The Pragmatic Programmers

TURING 图灵程序设计丛书

Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages

七周七语言

理解多种编程范型

[美] Bruce A. Tate 著

戴玮 白明 巨成 译



- 2011年Jolt大奖图书
- 带你轻松入门七种先锋语言
- 开阔视野，享受更多编程乐趣

人民邮电出版社

POSTS & TELECOM PRESS

作者简介

Bruce A. Tate

RapidRed公司总裁，该公司主要为Ruby轻量级开发提供咨询。他曾任职于IBM公司，并担任过多家公司的客户解决方案总监和CTO。著作有十余本，包括荣获Jolt大奖的*Better, Faster, Lighter Java*。

译者简介

戴玮

80后宅男，中科院自动化所在读博士，热爱机器学习与计算机视觉。编程喜C#、Ruby、Haskell而厌Java。

白明

Neusoft某开发部技术总监，拥有多年电信级软件开发经验；资深C程序员，对Common Lisp、Python等小众语言也有涉猎；对开源软件有着浓厚的兴趣，曾先后贡献了lcut、cbehave等多个工具框架；喜好写博客（http: //tonybai.com），耕耘七年，仍孜孜不倦。

巨成

万方数据数字出版事业部架构师，.NET开发人员。

图书在版编目（CIP）数据

七周七语言 : 理解多种编程范型 / (美) 泰特
(Tate,B.A.)著 ; 戴玮, 白明, 巨成译. -- 北京 : 人
民邮电出版社, 2012.5
(图灵程序设计丛书)
书名原文: Seven Languages in Seven Weeks: A
Pragmatic Guide to Learning Programming Languages
ISBN 978-7-115-27611-7

Ⅰ. ①七… Ⅱ. ①泰… ②戴… ③白… ④巨… Ⅲ.
①程序语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第037361号

内 容 提 要

本书共介绍了七种不同的编程语言。对于每种语言，分别介绍了各自的特性、应用，以及编程入门知识和关键编程范型，还带领读者使用能够代表该语言最重要特性的技术，解决某个不寻常的问题，使其充分掌握每种语言。

本书适合从事程序设计工作的人员阅读。

图灵程序设计丛书

七周七语言：理解多种编程范型

◆ 著　　[美] Bruce A. Tate
译　　戴玮　白明　巨成
责任编辑　朱　巍
执行编辑　丁晓昀

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：16.25
字数：390千字　　　　2012年5月第1版
印数：1－4 000册　　　　2012年5月北京第1次印刷

著作权合同登记号　图字：01-2011-7807号

ISBN 978-7-115-27611-7

定价：59.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

序　　言

选自未完成的《拥抱编程年华》①

——Erlang语言的作者Joe Armstrong

"Gmail编辑器不能正确排版引文格式。"

"这可够丢人的，"马杰里说，"看来他们负责引文格式的程序员水平不行，企业文化也在走下坡路。"

"我们该怎么办？"

"今后我们招募的程序员，一定得通读过《追忆似水年华》②。"

"读过全部七卷？"

"没错。"

"读这书能让人更好地驾驭标点符号，从而不至于在引文格式上犯错？"

"那倒未必，不过他们会因此拥有更精湛的编程技艺。这种感觉只可意会、不可言传……"

学编程就好比学游泳，再好的理论也不如一头扎下水，扑腾着呼吸新鲜空气管用。在初次没入水面的那一刻，你必定会惊慌失措，但当你奋力浮出水面、大口大口地喘着气，你又会无比喜悦。这时你心里明白："我学会游泳了。"至少我当初学游泳那会儿，就是这种感受。

编程也同样如此——迈出第一步最难。因此你需要一位好老师，鼓励你勇敢地跳入水中。

Bruce Tate正是这样的好老师。他写的这本书，带你从编程学习中最困难的地方入手，鼓励你大胆迈出第一步。

假设你想学习某门语言，而且顺利完成了下载安装编译器或解释器的艰巨任务，接下来要做

---

① 英文原文为*How Proust Can Make You a Better Programmer*（查不到任何相关信息，当为杜撰之作），直译应为《普鲁斯特如何让你成为更优秀的程序员》，但其书名模仿阿兰·德波顿（Alain de Botton）的《拥抱逝水年华》（*How Proust Can Change Your Life*），译为《拥抱编程年华》似更贴切。

②《追忆似水年华》（*À la recherche du temps perdu*），法国作家马塞尔·普鲁斯特所著小说，共七卷，细致刻画了19世纪末20世纪初的法国上流社会和其中的文人雅士们。不同于传统文学单纯地描写人类社会，该小说着重于分析人的心理情绪，首次在文学作品中成熟地剖析了人类情感，开"意识流"小说之先河，标志着文学史上一种新文学形式的诞生。Joe Armstrong为何借杜撰作品中的对话来表明，读《追忆似水年华》可提高编程水平？据本书作者Bruce Tate解释，此书让人更好地感受生活、理解生活、改变生活，自然也能让人成为更优秀的程序员。真是颇具禅意。

什么？你用它写的第一个程序，会是个什么样子？

Bruce回答得十分巧妙。他在这本书里，展示了许多完整程序和代码片段，你只需将它们一一输入，看看结果是否与书上相同。也就是说，你先不要想着自己编写程序，而是先把书中范例全都实现一遍。随着信心渐长，你会逐渐拥有独立完成编程项目的能力。

获得任何新技能的第一步，是先别想着独立解决什么，而是重复一遍前人已竟之事，这是掌握一门技能最快的方法。

用一门新语言上手编程的过程，与其说是投入大量时间反复实践，试图理解语言背后蕴含的深奥原理，还不如说是让分号、逗号各就各位，同时读懂出错时系统反馈的千奇百怪的错误信息。只有不断提高编程水平，让自己超越先输入代码、再等待编译成功的枯燥阶段，你才有能力思考程序语言中各种语法结构的含义。

跨过输入、运行程序的门槛后，你会有如释重负的感觉，因为潜意识将接管余下的工作。意识刚琢磨出分号放哪儿，潜意识就已明白了表面结构下的深层含义。这样下去，你终会有所顿悟，理解某个程序逻辑的更深层含义，某种语言结构如此特殊的原因，等等。

对几门语言均略知一二，这其实是一项相当实用的技能，因为我常常发现，网上的某个程序有助于解决手头问题，却没法直接拿来使用，还得针对问题稍作调整才行。这程序用什么语言写的都有可能，所以懂点儿Python、Ruby什么的就非常管用。

每门语言都自有一套惯用法。它们各有所长，亦各有所短。通过学习各种不同的编程语言，你会明白，哪门语言最适宜解决自己当下关注的问题。

Bruce对编程语言的爱好不拘一格，这真是你我之幸。他不仅精通那些声名卓著的语言，比如Ruby，还了解那些鲜为人知的语言，比如Io。编程说到底是个理解问题，理解说到底又是个思想问题，因此，若想深入理解编程的方方面面，洞察新近涌现的思想是必不可少的一环。

精于禅宗的大师会告诉你，拉丁语学得越好，数学也就越好。编程也同样如此。通过研究逻辑式编程或函数式编程，你能领悟到面向对象编程的精华；通过学习汇编语言，你能更透彻地理解函数式编程。

在我做程序员时，对比各门编程语言的书籍曾一度盛行。这些大部头书多带有学术腔调，至于如何去真正用好哪门语言，则少有涉及。这如实反映了那个年代的技术发展水平。当时，我们只能从书本上了解某门语言的诸般理念，想用它实战几乎不太可能。

如今，我们不仅能了解这些理念，还能对它们实践一番。伫立池畔、畅想游泳之妙，较之亲身跳入水中、畅享戏水之乐，二者终究不可同日而语。

我由衷地推荐这本书。也希望你在阅读它的时候，能够像我一样，尽情享受其中的乐趣。

Joe Armstrong，Erlang语言之父
2010年3月2日
于斯德哥尔摩

致　　谢

这是我付出最多的一本书，但回报也同样最为丰厚。在大家为我提供的各种帮助下，我才得以顺利写完此书。首先，我一定要感谢我的家人。Kayla和Julia，你们的写作让我惊喜。你们今后定能达到超乎自己想象的高度。Maggie，你是我的开心果，也是我的灵感源泉。

感谢Ruby社区中的Dave Thomas，是你带我来到这门彻底改变我职业生涯、让我重拾编程乐趣的语言面前。还要感谢Matz[①]，你的友情分享让读者有幸欣赏到你的真知灼见。你邀请我访问日本，让我有机会亲自拜访Ruby诞生之地，这段经历对我写作的启发是你难以想象的。此外，我要感谢的人还包括Charles Nutter[②]、Evan Phoenix[③]、Tim Bray[④]，感谢你们与我就书中话题交换意见。我们交谈的内容或许索然无味，但对我提炼和加工“Ruby一章”要点却着实大有助益。

感谢Io社区中的Jeremy Tregunna[⑤]带我入门，并在本书中分享了一些精妙范例。你所做的审阅工作也极其出色，不仅反馈及时，而且令本章内容更加充实有力。还要感谢Steve Dekorte[⑥]，无论编程语言市场认同与否，你都创造了一门超凡的语言。它的并发特性激动人心，其自身也散发着与生俱来的魅力。在我眼中，它无疑是一门极优秀的语言。谢谢你帮我这只菜鸟搞定了安装问题，也谢谢你的精心审阅，还有那帮我理解了Io本质的访谈。你俘虏了提前试读本书的读者的心，让Io成为他们最喜爱的语言。

感谢Prolog社区中的Brian Tarbox[⑦]与读者分享你那非同凡响的经历。你的海豚研究项目（某一期*Nova*[⑧]曾以此为内容）为这一章增添了生动有趣的情节。特别要感谢Joe Armstrong，你的反馈不仅对“Prolog一章”、而且对全书的形成发展都有着莫大帮助。还要感谢你提供的地图着色的例子以及对Append的见解，它们都是在恰当时间出现的恰当示例。

---

① 松本行弘（Mast umoto Yukihiro），Ruby语言之父，其探索程序设计思想和方法的经典作品《松本行弘的程序世界》已由人民邮电出版社出版。——编者注

② JRuby的核心开发者之一。（以下注释若无特殊说明，均为译者注。）

③ Rubinius（Ruby语言的一种实现）的发明者。

④ 曾任Sun公司Web技术总监，现任Google公司开发大使（Developer Advocate），主要关注Android。同时，他还是多家公司的联合创始人。

⑤ Io语言的核心开发者之一。

⑥ Io语言的发明者，同时也是核心开发者之一。

⑦ 摩托罗拉公司系统工程组的杰出工程师，自PDP-11计算机开始编程，经验极为丰富。

⑧ 一档美国剧集的名字，由公共电视网（Public Broadcasting Service, PBS）播出。每一期节目都会采访一位科学家，并以该科学家的研究工作为主题进行探讨。其节目富有科学气息，颇受美国乃至世界各地观众的欢迎。

感谢Scala社区中我的好友Venkat Subramaniam[①]。你写的那本Scala书既通俗又有料，连我都不禁为之倾倒。十分感激你的审阅、以及审阅后提出的些许建议。这些建议于你只是举手之劳，于我却减轻了极大负担，让我可以把精力集中在传道授业的本原上。还要感谢Martin Odersky[②]，我们之前虽素昧平生，但你仍欣然与本书读者分享观点。Scala走了一条与众不同而荆棘丛生之路，因为它力图把函数式编程和面向对象两种范型合二为一。你们为此所做的努力，我们都看在眼里、记在心上。

再次感谢Erlang社区中的Joe Armstrong。你的友善与活力，帮我理清了初写此书时错综复杂的思路。你不知疲倦地推广这一理念：系统应当以分布式、容错的方式构建出来。现今推广工作已初见成效。和本书其他语言的任何概念都不大一样，我觉得Erlang"就让它崩溃"的哲学特别实用。希望能看到Erlang的这些思想越来越多地应用于实践。

感谢Clojure社区中的Stuart Halloway[③]，你的审阅意见督促我加倍努力，让这本书精益求精。你对Clojure的深刻理解与独特直觉，将其精要之处一一展现在了我的面前。你写的那本书也深深影响了Clojure这章，甚至切实改变了我处理其他章节问题的方式。我也十分赞赏你在咨询业的工作方法，是你把该行业迫切需求的简洁和高效引入进来。我还要感谢Rich Hickey[④]，你让我了解到这门语言如何诞生、以及它为什么是一门Lisp方言。你的某些思想虽颇为极端，却相当实用。祝贺你，你又发现了一条Lisp的革新之路[⑤]。

感谢Haskell社区中的Phillip Wadler[⑥]，你让我有机会深入了解Haskell的诞生过程。在我们交流了传授知识的感受之后，我发现你真是这方面的行家里手。还要感谢Simon Peyton-Jones[⑦]，能把你的访谈、深刻见解和独到观点带给读者，对我而言无疑是一大乐事。

本书的审阅者们非常漂亮地完成了审阅任务。这里，我要感谢Vladimir G. Ivanovic、Craig Riecke、Paul Butcher、Fred Daoud、Aaron Bedra、David Eisinger、Antonio Cangiano、Brian Tarbox，你们组成了我合作过的最有力的一支审阅团队。有了你们，这本书才能如此出色。我知道，逐字逐句地仔细审阅一本书需要耗费极大精力，也是回报与付出不成正比的一项工作。那些一如既往热爱技术图书的人们都会对你们心存感激。若没有你们，出版业将不复存在。

我还要感谢那些分享语言偏好和编程哲学方面见解的人。在我写书的不同阶段，Neal Ford、John Heintz、Mike Perham、Ian Warshak都做了重要贡献。与他们谈天说地让我获益良多，也让

① 多本技术书的作者，包括《Scala程序设计：Java虚拟机多核编程实战》、《高效程序员的45个习惯：敏捷开发修炼之道》等。

② Scala语言的发明者，并建立了支持和推广Scala的公司Typesafe。

③ *Programming Clojure*一书的作者。

④ Clojure语言的发明者。

⑤ 为什么要说"又"呢？实际上，Rich Hickey在发明Clojure之前，还开发过一个类似于Clojure但位于.Net平台上的项目dotLisp。

⑥ 美国计算机科学家，爱丁堡大学理论计算机科学教授，Haskell语言的设计者之一。他在编程语言设计和类型论等领域功勋卓著，包括建立函数式编程的基本理论、将单子（monad）引入函数式编程、设计Haskell语言等。

⑦ 英国计算机科学家，格拉斯哥大学荣誉教授，也在剑桥大学指导博士生。他是Haskell的设计者之一，将惰性求值（lazy evaluation）机制引入Haskell，并领导开发了著名的格拉斯哥Haskell编译器（Glasgow Haskell Compiler，GHC）。

我在书里的表现比我真实水平高了不少。

还有试读者，谢谢你们阅读本书，并鞭策我不断前进。你们的评论让我感觉到，你们有不少人真是在用心学这些语言，而不只是草草浏览一遍了事。我已根据你们的评论多次修改了本书，而且我希望在本书的整个生命周期当中，这样的修改多多益善。

最后，我要向“Pragmatic Bookshelf丛书”团队致以最诚挚的感谢。Dave Thomas和Andy Hunt，你们二位作为程序员也好，作为技术书作者也好，都对我的职业生涯有不可估量的巨大影响。你们的出版平台，为我再一次提供了写作机会，让这样一本对大众市场来说未必有太大吸引力的书问世，而且还能卖得不错。谢谢出版团队的全体成员。Jackie Carter，你的友善帮助与指导是本书不可或缺的，我非常享受我们之间的每次交谈，但愿你也有同样感受。感谢那些在幕后默默工作的人们，你们让这本书做到了最棒。具体说来，我要感谢文字编辑Kim Wimpsett、索引编辑Seth Maislin、排版编辑Steve Peter、印刷编辑Janet Furlow，是你们的辛勤工作让这本书如此优秀。没有你们，这本书决不会像现在这样出色。

当然，全部错误都由我负责，与这个优秀的出版团队无关。如果有什么遗漏之处，我愿致以最衷心的歉意。任何疏忽都不是有心之过。

最后，谢谢我所有的读者。因为有你们读我的书，我才觉得这些印刷出来的一页页书不是一堆废纸，我的写作热情也才会不可抑制地喷薄而出。

Bruce A. Tate

目　　录

第 1 章　简介 …… 1
1.1　不走寻常路 …… 1
1.2　语言 …… 2
1.3　谁应该买这本书 …… 4
1.3.1　学会如何学习 …… 4
1.3.2　乱世英雄 …… 4
1.4　谁不应该买这本书 …… 5
1.4.1　超越语法 …… 5
1.4.2　不是安装指南 …… 6
1.4.3　不是编程参考 …… 6
1.4.4　严格督促 …… 6
1.5　最后一击 …… 7

第 2 章　Ruby …… 8
2.1　Ruby 简史 …… 9
2.2　第一天：找个保姆 …… 10
2.2.1　快速起步 …… 10
2.2.2　从命令行执行 Ruby …… 10
2.2.3　Ruby 的编程模型 …… 11
2.2.4　判断 …… 12
2.2.5　鸭子类型 …… 15
2.2.6　第一天我们学到了什么 …… 16
2.2.7　第一天自习 …… 16
2.3　第二天：从天而降 …… 17
2.3.1　定义函数 …… 17
2.3.2　数组 …… 17
2.3.3　散列表 …… 19
2.3.4　代码块和 yield …… 20
2.3.5　定义类 …… 22
2.3.6　编写 Mixin …… 25
2.3.7　模块、可枚举和集合 …… 26
2.3.8　第二天我们学到了什么 …… 27
2.3.9　第二天自习 …… 28
2.4　第三天：重大改变 …… 28
2.4.1　打开类 …… 29
2.4.2　使用 method_missing …… 30
2.4.3　模块 …… 31
2.4.4　第三天我们学到了什么 …… 35
2.4.5　第三天自习 …… 35
2.5　趁热打铁 …… 35
2.5.1　核心优势 …… 35
2.5.2　不足之处 …… 37
2.5.3　最后思考 …… 37

第 3 章　Io …… 38
3.1　Io 简介 …… 38
3.2　第一天：逃学吧，轻松一下 …… 39
3.2.1　开场白 …… 39
3.2.2　对象、原型和继承 …… 40
3.2.3　方法 …… 43
3.2.4　列表和映射 …… 44
3.2.5　true、false、nil 以及单例 …… 46
3.2.6　Steve Dekorte 访谈录 …… 48
3.2.7　第一天我们学到了什么 …… 49
3.2.8　第一天自习 …… 50

3.3 第二天：香肠大王 50
3.3.1 条件和循环 51
3.3.2 运算符 52
3.3.3 消息 54
3.3.4 反射 56
3.3.5 第二天我们学到了什么 57
3.3.6 第二天自习 57
3.4 第三天：花车游行和各种奇妙经历 58
3.4.1 领域特定语言 58
3.4.2 Io 的 method_missing 60
3.4.3 并发 62
3.4.4 第三天我们学到了什么 64
3.4.5 第三天自习 64
3.5 趁热打铁 65
3.5.1 核心优势 65
3.5.2 不足之处 66
3.5.3 最后思考 66

第 4 章 Prolog 67
4.1 关于 Prolog 68
4.2 第一天：一名优秀的司机 68
4.2.1 基本概况 68
4.2.2 基本推论和变量 69
4.2.3 填空 70
4.2.4 合一，第一部分 73
4.2.5 实际应用中的 Prolog 75
4.2.6 第一天我们学到了什么 77
4.2.7 第一天自习 77
4.3 第二天：离瓦普纳法官开演还有15 分钟 78
4.3.1 递归 78
4.3.2 列表和元组 79
4.3.3 列表与数学运算 82
4.3.4 在两个方向上使用规则 83
4.3.5 第二天我们学到了什么 86
4.3.6 第二天自习 87
4.4 第三天：维加斯的爆发 87
4.4.1 解决数独问题 87
4.4.2 八皇后问题 91
4.4.3 第三天我们学到了什么 96
4.4.4 第三天自习 96
4.5 趁热打铁 96
4.5.1 核心优势 97
4.5.2 不足之处 97
4.5.3 最后思考 98

第 5 章 Scala 99
5.1 关于 Scala 99
5.1.1 与 Java 的密切关系 100
5.1.2 没有盲目崇拜 100
5.1.3 Martin Odersky 访谈录 100
5.1.4 函数式编程与并发 101
5.2 第一天：山丘上的城堡 102
5.2.1 Scala 类型 102
5.2.2 表达式与条件 103
5.2.3 循环 105
5.2.4 范围与元组 107
5.2.5 Scala 中的类 109
5.2.6 扩展类 111
5.2.7 第一天我们学到了什么 113
5.2.8 第一天自习 114
5.3 第二天：修剪灌木丛和其他新把戏 114
5.3.1 对比 var 和 val 115
5.3.2 集合 115
5.3.3 集合与函数 119
5.3.4 第二天我们都学到了什么 124
5.3.5 第二天自习 124
5.4 第三天：剪断绒毛 125
5.4.1 XML 125
5.4.2 模式匹配 126
5.4.3 并发 128
5.4.4 实际中的并发 129

5.4.5　第三天我们学到了什么 …… 132
5.4.6　第三天自习 …… 132
5.5　趁热打铁 …… 133
5.5.1　核心优势 …… 133
5.5.2　不足之处 …… 134
5.5.3　最后思考 …… 135

第 6 章　Erlang …… 136

6.1　Erlang 简介 …… 136
6.1.1　为并发量身打造 …… 137
6.1.2　Joe Armstrong 博士访谈录 …… 138
6.2　第一天：以常人面目出现 …… 139
6.2.1　新手上路 …… 140
6.2.2　注释、变量和表达式 …… 140
6.2.3　原子、列表和元组 …… 141
6.2.4　模式匹配 …… 142
6.2.5　函数 …… 144
6.2.6　第一天我们学到了什么 …… 147
6.2.7　第一天自习 …… 147
6.3　第二天：改变结构 …… 148
6.3.1　控制结构 …… 148
6.3.2　匿名函数 …… 150
6.3.3　列表和高阶函数 …… 150
6.3.4　列表的一些高级概念 …… 153
6.3.5　第二天我们学到了什么 …… 156
6.3.6　第二天自习 …… 157
6.4　第三天：红药丸 …… 157
6.4.1　基本并发原语 …… 157
6.4.2　同步消息 …… 160
6.4.3　链接进程以获得可靠性 …… 162
6.4.4　第三天我们学到了什么 …… 167
6.4.5　第三天自习 …… 167
6.5　趁热打铁 …… 167
6.5.1　核心优势 …… 168
6.5.2　不足之处 …… 169
6.5.3　最后思考 …… 169

第 7 章　Clojure …… 170

7.1　Clojure 入门 …… 170
7.1.1　一切皆 Lisp …… 170
7.1.2　JVM …… 171
7.1.3　为并发更新 …… 171
7.2　第一天：训练 Luke …… 172
7.2.1　调用基本函数 …… 172
7.2.2　字符串和字符 …… 174
7.2.3　布尔值和表达式 …… 175
7.2.4　列表、映射表、集合以及向量 …… 176
7.2.5　定义函数 …… 179
7.2.6　绑定 …… 180
7.2.7　匿名函数 …… 182
7.2.8　Rich Hickey 访谈录 …… 183
7.2.9　第一天我们学到了什么 …… 184
7.2.10　第一天自习 …… 185
7.3　第二天：Yoda 与原力 …… 185
7.3.1　用 `loop` 和 `recur` 递归 …… 185
7.3.2　序列 …… 186
7.3.3　延迟计算 …… 189
7.3.4　`defrecord` 和 `protocol` …… 191
7.3.5　宏 …… 193
7.3.6　第二天我们学到了什么 …… 194
7.3.7　第二天自习 …… 195
7.4　第三天：一瞥魔鬼 …… 195
7.4.1　引用和事务内存 …… 196
7.4.2　使用原子 …… 197
7.4.3　使用代理 …… 198
7.4.4　future …… 200
7.4.5　还差什么 …… 200
7.4.6　第三天我们学到了什么 …… 201
7.4.7　第三天自习 …… 201
7.5　趁热打铁 …… 201
7.5.1　Lisp 悖论 …… 202

7.5.2 核心优势 202
7.5.3 不足之处 203
7.5.4 最后思考 204

第 8 章 Haskell 205

8.1 Haskell 简介 205
8.2 第一天：逻辑 206
8.2.1 表达式和基本类型 206
8.2.2 函数 208
8.2.3 元组和列表 211
8.2.4 生成列表 214
8.2.5 Philip Wadler 访谈录 217
8.2.6 第一天我们学到了什么 218
8.2.7 第一天自习 218
8.3 第二天：Spock 的超凡力量 218
8.3.1 高阶函数 219
8.3.2 偏应用函数和柯里化 220
8.3.3 惰性求值 221
8.3.4 Simon Peyton-Jones 访谈录 223
8.3.5 第二天我们学到了什么 224
8.3.6 第二天自习 225
8.4 第三天：心灵融合 226
8.4.1 类与类型 226
8.4.2 monad 231
8.4.3 第三天我们学到了什么 236
8.4.4 第三天自习 237
8.5 趁热打铁 237
8.5.1 核心优势 237
8.5.2 不足之处 238
8.5.3 最后思考 239

第 9 章 落幕时分 240

9.1 编程模型 240
9.1.1 面向对象（Ruby、Scala） 241
9.1.2 原型编程（Io） 241
9.1.3 约束-逻辑编程（Prolog） 241
9.1.4 函数式编程（Scala、Erlang、Clojure、Haskell） 241
9.1.5 范型演进之路 242
9.2 并发 242
9.2.1 控制可变状态 243
9.2.2 Io、Erlang 和 Scala 中的 actor 243
9.2.3 future 243
9.2.4 事务型内存 243
9.3 编程结构 244
9.3.1 列表解析 244
9.3.2 monad 244
9.3.3 匹配 245
9.3.4 合一 246
9.4 发现自己的旋律 246

附录 参考书目 247

第 1 章 简介

人们出于各种目的学习自然语言。学母语是为了生存，为了日常生活中与人正常交往。学外语的目的可就五花八门了。有时候，为了未来的职业发展或为了适应日益变化的生活环境，你不得不学习外语；但有时候，你决心征服一门外语，不是因为不得不这么做，而是因为发自内心地想学。外语能带你领略一片未曾见过的风景。你甚至可能领悟一个道理：每学一门新的语言，思维方式都会发生改变。

编程语言亦是如此。在这本书中，我将为你介绍七门各不相同的语言。不过，我不会像你的妈妈那样将吃的直接喂到你嘴边。我更愿意做你的导游，带你体验一次启迪心智之旅，并由此改变你看待编程的视角。写这书的目的不是让你成为专家，而是教会你比“Hello, World”更实用的知识。

1.1 不走寻常路

假如我想新学一门编程语言或一种编程框架，一般会找一篇速成互动教程看看。因为这类教程中，先做什么、后做什么都已精心设计好。通过它们，我们可以更容易体会语言的妙处所在。当然，扔掉教程，直接动手实践也未尝不可，但说白了，我就是想尽快发现语言的动人心弦之处，尽快对它的语法糖[①]和核心概念有个大体印象。

然而多数情况下，我找不到称心如意的教程。受到篇幅限制，那些教程往往只介绍各门语言间相去无几的皮毛。而这些皮毛，我又早已熟知。若想领会一门语言的精髓，它可就无能为力了。我想要的是那种痛快淋漓、深入探索语言本质的感觉。

本书将会给你这种感觉。不是一次，而是七次。你将从书中找到以下问题的答案。

- 语言的类型模型是什么？强类型（Java）或弱类型（C语言），静态类型（Java）或动态类型（Ruby）。本书侧重于介绍强类型语言，但各种静态类型和动态类型语言也都有所涉及。你将看到，语言在类型模型间的权衡会对开发者产生何种影响。语言的类型模型会改变你对问题的处理方式，还会控制语言的运行方式。就类型模型而言，书中的每门语言都

① 术语“语法糖”是由英国计算机科学家Peter Landin首次提出的。该术语指的是为编程语言添加某种语法，这种语法虽对语言本身功能并无实质影响，但为程序员编程提供了便利，比如C语言用`a[i]`表示`*(a+i)`、用`for`表示循环等语法。语法糖是本书重点关注的主题之一。

堪称独树一帜。

- 语言的编程范型是什么？是面向对象（object-oriented，OO）、函数式、过程式，还是它们的综合体？本书介绍的语言涵盖了4种编程范型，有些语言还由几种范型组合而成。你将看到一门基于逻辑的编程语言（Prolog）、两门完全支持面向对象思想的语言（Ruby和Scala）、四门带有函数式特性的语言（Scala、Erlang、Clojure和Haskell）及一门原型语言（Io）。这里有Scala这样的多范型（multiparadigm）语言，也有Clojure这种多方法（multimethod）语言，后者甚至允许你实现自定义范型。本书最重要的任务之一，就是学习新的编程范型。
- 怎样和语言交互？语言可编译也可解释，可以有虚拟机也可以没有。在本书中，如果某门语言带交互命令行，将先通过交互命令行探索这门语言，当我们处理规模较大的项目时，还会转而采用文件编程。我们接触的项目不会特别大，因此无需深入研究打包（packaging）模型。
- 语言的判断结构（decision construct）和核心数据结构是什么？或许你会惊讶，在作判断时，居然如此多的语言都用到了与`if`和`while`的各种变型都不相同的结构。你会见识到Erlang的模式匹配，还有Prolog的合一（unification）①。至于数据结构，集合（collection）在任何语言中都扮演着至关重要的角色。对Smalltalk和Lisp这类语言，集合刻画了语言特征，而在C++和Java等语言中，集合更可谓无所不在，它们决定着用户体验，若没了它们，语言势必成为一盘散沙。因此，无论用哪一类语言，都必须全面、透彻地理解集合。
- 哪些核心特性让这门语言与众不同？有些语言支持并发编程的高级特性，有些语言提供独一无二的高级结构，比如Clojure的宏（marco）和Io的消息解释（message interpretation）；有些语言包含性能强劲的虚拟机，如Erlang的BEAM，它能让 Erlang构建的容错分布式系统远远快于其他语言；有些语言提供专门针对特定问题的编程模型，比如利用逻辑规则解决约束问题。

就算这些问题全被你弄个一清二楚，你仍然成不了语言专家，哪怕只是其中一门语言。但你会明白，这几门语言各自拥有哪些独门绝技。下面，我们先看看本书介绍了哪几门语言。

1.2　语言

从众多语言中，挑出本书包含的几门语言，这一过程也许不像你想得那么复杂。我们只不过发了些调查问卷，向本书的潜在读者请教了一番。调查数据汇总上来时，有八门语言入选希望最大。不过，我先把JavaScript“踢”了出去，因为它实在是过于热门了，取而代之的是原型语言中热门程度仅次于JavaScript的Io。随后，我又把Python“踢”了出去，因为我只想给面向对象语言一个名额，而Ruby的票数多于Python。同时，这也给一个出人意料的候选者让出了位置——名单

① 合一是数理逻辑中的概念，指的是找到某个置换，使得两个项（term）完全一致。它也是Prolog的核心思想之一。Prolog中的合一包括原子和原子的合一，原子、项或另一未实例化变量和未实例化变量的合一，项和项的合一等三种情况，本书4.2节、4.3节对此有详细说明。

上位列前十的Prolog。下面，我给出成功入围本书的最终名单以及挑选它们的理由[①]。

- Ruby。这门面向对象语言高票当选，因为它不仅好用，而且好读。我曾经考虑过不介绍任何一门面向对象语言，但我又想在其他编程范型与面向对象编程之间作一些比较，因此，至少介绍一门面向对象语言还是有必要的。相比于大多数程序员的日常用法，我想把它挖掘得更深入一些，以揭示设计者的良苦用心。我最终决定重点介绍Ruby元编程（metaprogramming），因为它可以用来扩展Ruby的语法。对于Ruby榜上有名的结果，我还是相当认可的。
- Io。和Prolog一样，Io也是本书颇具争议的语言。它虽与商业成功无缘，但其兼具简单性和语法一致性的并发结构，却是十分重要的思想。它的最简语法（minimal syntax）功能强大，与Lisp的相似性也颇能给人留下几分印象。Io不仅和JavaScript一样同为原型语言，还有着独一无二、韵味无穷的消息分发机制，因此在众多编程语言之中，它也占有小小的一席之地。
- Prolog。没错，Prolog年事已高，但它仍然威力无穷。它能轻松解出数独问题，这着实让我大开眼界。用Java或C语言时，有些难题我殚精竭虑方能解决，用Prolog却能干净利落地搞定。承蒙Erlang发明者Joe Armstrong出手相助，我得以深刻体会到Prolog之妙，而且也正是深受Prolog影响，Erlang才得以问世。如果你此前从未用过Prolog，我保证，它定会带给你惊喜。
- Scala。作为运行于Java虚拟机上的新一代语言，Scala为Java系统引入了强大的函数式思想，同时也并未丢弃面向对象编程。回顾历史，我发现C++和Scala有着惊人的相似之处，因为从过程式编程过渡到面向对象编程期间，C++同样起到了举足轻重的作用。当你真正融入Scala社区之后，你就会明白，为什么对于函数式语言程序员来说，Scala是异端邪说，而对于Java开发者来说，Scala是天降福音。
- Erlang。作为名单上历史最悠久的语言之一，Erlang不仅是一门函数式语言，而且在并发、分布式编程、容错等诸多方面都有优异表现，真是想不火都难。CouchDB（新兴的基于云的数据库）的创始人就选择了Erlang，并且义无反顾地一直用它，只要花上点时间了解这门分布式语言，你就会明白原因所在。在Erlang帮助下，设计带有并发、分布式、容错等特征的应用程序将变得无比简单。
- Clojure。这又是一门Java虚拟机语言，但正是这门Lisp方言，彻底颠覆了我们在Java虚拟机上并发编程的思考方式。它是本书唯一在版本数据库中使用同一种策略管理并发的语言。作为Lisp方言，Clojure或许拥有本书所有语言中最灵活的编程模型，因此绝不缺乏号召力。与其他Lisp方言不同的是，它不会带那么多括号[②]，还有众多Java库和在各平台上的广泛部署作为坚强后盾。

① 票数最高的八门候选语言是：Ruby、Python、JavaScript、Haskell、Scala、Erlang、Clojure、Prolog。由于正文所述的理由，作者最终挑选了本书包含的这七门语言。至于为什么是七门语言，而不是六门或八门，据作者解释，首先，主要考虑哪些语言最适合读者，斟酌之后，确定下来的语言恰好就是这七门；另外，西方文化视“7”为幸运数字，选择七门语言与此也有一定关系，当然，这绝非主要因素。

② 关于Lisp语言的括号，有一个广为流传的笑话：据说，一个黑客冒死偷到了美国用于导弹控制的Lisp代码的最后一页，却发现那一页上全是右括号“)”。

- Haskell。它是本书唯一的纯函数式语言，这也意味着，它根本不存在可变状态：只要使用相同的输入参数，去调用相同的函数，就会返回相同的输出。在所有强类型语言中，Haskell拥有最令人称羡的类型模型。和Prolog一样，它也需要你花一些时间理解，但你得到的回报绝对物超所值。

如果名单上没有你钟爱的语言，我深感抱歉。老实说，还真有语言狂热分子给我发过好几封恐吓信。在本节开始提到的民意调查中，我们总共列出了几十门语言。我挑的这几门语言未必是其中最出色的，但它们特点突出、个性鲜明，都具有重要的学习价值。

1.3　谁应该买这本书

如果你是一名称职的程序员，想提高自己的编程水平，那你应该买这本书。这话说来有几分含糊，请容我解释一二。

1.3.1　学会如何学习

Dave Thomas是Pragmatic Bookshelf出版社的创始人之一，我这本书就是他们出版的。他每年都鼓励数以千计的学生去学一门新语言。学过各式各样的语言后，你最少也能挑出一门得心应手的语言用用，并把其他语言的精华思想融入到这门语言的代码中去。

这本书的写作过程已经深刻影响了我所编写的Ruby代码。相比于过去，我编写的Ruby代码中，函数式味道更加浓郁，且因重复部分变少而增加了可读性。我在代码中尽量缩减了可变变量的数量，还利用代码块和高阶函数写出了更有效的代码。此外，我也用到一些不大符合Ruby惯例，但会让代码更简明的技巧。

学语言最理想的情况，是由它引领你踏上一条崭新的职业道路。每十年左右，编程范型都会发生一次变革。几年前，我感觉Java越来越别扭，于是就去体验了一把Ruby，看看怎么用它进行Web开发。经过几个过渡项目的磨合，我开始重点发展Ruby方向上的业务，从此彻底告别Java。我的Ruby生涯始于玩票，但随之而来的，却是事业的不断发展壮大。

1.3.2　乱世英雄

说到本书读者，他们大概还没那么老，不至于经历过上一次编程范型的更新换代。回想刚换到面向对象编程那会儿，我们遇到过好几次挫折，不过话说回来，当时的结构化编程范型已完全无法应付现代Web应用的复杂性。Java编程语言的成功为Web应用开发打了一针强心剂，也因此奠定了面向对象编程这种新编程范型的地位。不过，当时很多开发者已深深陷入了过时技术的桎梏中。他们若想顺利过渡到新编程范型，必须由内到外重新打造思考编程的方式、手头用于开发的工具、设计应用程序的方法等才行。

现在，我们可能正身处又一次变革的进程当中。这一次变革，新的计算机设计架构将成为主要推动力。在本书的七门语言中，五门都拥有强大的并发模型（Ruby和Prolog不在其列）。无论

你用的编程语言会不会一夜之间物是人非，我敢向你保证，在应对这场变革之时，本书介绍的所有语言都能拿出令人信服的策略。看看Io对future的实现、Scala的actor、Erlang的“任其崩溃”（let it crash）哲学，再看看Haskell如何把可变状态抛到九霄云外、Clojure如何利用版本控制解决最为棘手的并发问题，你就会相信这一点。

当然，那些看似平凡的语言也不可小觑，它们带来的启示同样让人啧啧称奇。Erlang这门用于多个云数据库后台的语言就是个极佳的例子。正是以Prolog为基础，Joe Armstrong博士创立了这门语言[①]。

1.4　谁不应该买这本书

如果你没有读过本节，或读过但不认同其中观点，那你不应该买这本书。买这本书等于跟我做了笔买卖：你认可我把重点放在编程语言本身而非详尽的安装过程上，我承诺在有限时间内尽可能多地授业解惑。你要学会利用Google搜索那些细枝末节，可别指望我会帮你解决各种安装问题。如此一来，我才有空间深入挖掘语言本身，而你在读过本书后，也才能了解更多语言方面的细节。

请务必明白，这七门语言，无论教还是学，对我们而言都是一个宏伟目标。作为读者，你的脑袋必须多腾出点地方，以容纳七种不同的语法风格、四种编程范型、四十年语言开发的宝贵经验；作为作者，我必须尽量全面地涵盖各个主题，以便让你更好地理解语言。为了写好这本书，我老早就学过了这七门语言中的几门，但若想完美地兼顾每门语言所有最重要的细节，还需要一些化繁为简的本事才行。

1.4.1　超越语法

想真正理解语言设计者的思路，就必须有超越基本语法的觉悟。这意味着，你不能仅仅停留在编写“Hello, World”这种普通代码，甚至斐波那契数列代码的水平。如果是Ruby，你得会写一些元编程代码；如果是Prolog，你必须会解决完整的数独问题；如果是Erlang，你要懂得如何写一个监控程序，这程序不仅能检测崩溃进程，还能启动另一进程以接替崩溃进程的工作，或将崩溃进程的相关信息告知用户。

在地带你超越语法之前，我要先向你作个承诺，同时也不得不作个让步。承诺是：决不会浅尝辄止、敷衍了事；让步是：无法像专业语言书籍那样涵盖所有基础知识。我几乎没有涉及异常处理，除非它是哪一门语言的基本特性；我也没有详细介绍包模型，因为我们做的都是小项目，没有必要用到打包模型；还有，不少原始类型（primitive）我也只字未提，因为解决本书提出的基本问题时，用不到的原始类型自然不必提到。

① 这里的潜台词是：Prolog根本不支持任何并发特性，但以它为基础的Erlang，却是一门将并发作为招牌特性的语言。这足以说明语言的本质是什么，更提醒我们千万不要忽视任何一门语言。

1.4.2　不是安装指南

写这书最大的挑战来自于平台。我与各种书的不少读者都有过直接接触，他们所用的平台包括三种Windows平台、OS X以及至少五种Unix系统。我也在各大留言板上看过数不胜数的平台之争。把七门语言安装到七种平台上，这别说一位作者，就算多位作者合著，估计也是无解难题。我无意解决七门语言的安装问题，所以就不费那精力去琢磨多平台了。

我猜你不会有兴趣读一份老掉牙的安装指南。语言和平台都在不断发生变化。我只要告诉你去哪里安装语言、我用的是什么版本就够了。这样你就可以和大家一样，照着最新的安装指南去做。一步步地教你安装语言真没什么必要。

1.4.3　不是编程参考

为保证本书质量，我们尽最大努力对书中代码进行了审阅，其中一部分还有幸请到了语言设计者亲自审阅。在经历出版前的层层严格审阅之后，我确信，这些代码足以深刻阐释每一门语言的精髓。不过，当你自己试着上手用这七门语言编程时，我再怎么玩命，也不可能把一份全面的语言参考摆在你面前。请你多多谅解。关于这点，我想拿平时会话所用的语言打个比方。

观光旅游时学到的语言，和作为母语而熟知的语言相去甚远。我英语说得流畅自然，西班牙语却磕磕绊绊。还有三门语言，我也会说若干短语。我能在日本吃饭时点鱼，也能在意大利问人找洗手间。但我心知肚明的是，自己非母语的表达能力实在有限。说到编程，我的BASIC、C、C++、Java、C#、JavaScript、Ruby等几门语言都十分熟练。不甚熟练的语言也不少，其中还包括本书介绍的几门语言。说老实话，以我现在的水平，七门语言中有六门都不是非常得心应手。近五年当中，我一直全职编写Ruby代码，但说到其他语言，我是既说不出怎么用Io编个Web服务器，也说不出如何用Erlang编个数据库。

如果真去写一本这七门语言的参考大全，那我一定死得很惨。就算从中随便挑一门语言写编程指南，也至少会有咱们这本书差不多厚。我能提供各种材料，帮你轻松入门；也能带你体验每门语言的真实范例，让你亲眼见识它们的程序代码；还能尽量编译所有代码，确保它们正常运行。但如果你在试验这些语言时，也希望我能提供指导，那我真是心有余而力不足。

这七门语言都有非常优秀的支持社区，这也是我选择它们的原因之一。而且在每个习题环节，我还尽量保留了一个搜索语言相关资源的问题。用意很明显——让你学会自力更生。

1.4.4　严格督促

本书为你铺就的学习途径，较之网上那些20分钟教程可谓略胜一筹。我知道，你我同为善用Google之人，随便搜索书中某门语言的简明教程自是不在话下。不过本书的高明之处在于，它会带你踏上快速成长的互动之旅。你每周都会遇到一些小型的编程挑战和一个实战项目。解决它们虽非易事，但这既能增长你的见识，还可让你体验编程之乐。

如果你阅读本书时不做任何习题，那不过是对语法有了个粗浅认识。如果你在尝试独立解答

习题之前，先去网上搜索答案，那也一样意味着不及格。你首先要有试着解答习题的主观愿望，同时也要充分认识到，有一小部分习题可能超出了你的能力范围。要知道，学会语法永远比学思考简单。

如果以上描述让你心惊胆战，我建议你放下这本书，换本别的书看看。对你来说，也许看七本不同的编程语言书会更轻松惬意。但是，如果你马上想到的是看这本书所能带来的回报——写出一手更漂亮的代码——并为此激动不已，那就别犹豫了，赶紧往下看吧。

1.5　最后一击

此时此刻，我真想对你说几句意义深远又让人热血沸腾的话，但千言万语汇成四个字——享受编程。

第 2 章 Ruby

有糖相伴好下药。

——Mary Poppins

如果你正信手翻阅此书，那我们大概志趣相投——都喜欢学习各种编程语言。对我来说学一种语言，就如同了解一个人的性格一般。自我进入编程行业以来，用过的语言已不在少数，深知语言如人，每种语言都有其独特个性。Java像一位地主家的孩子，小时候天真可爱，但长大后开始巧取豪夺，方圆百里之内听不到一丝欢声笑语；Visual Basic像一位浓妆艳抹的美发师，虽对全球变暖问题一无所知，理发却是一把好手，言谈风趣幽默总能把人逗得开怀大笑。在本书中，我将把你学到的每门语言都比作某个著名的影视人物。希望这样的比喻能对你有所启发，让你多少明白各语言与众不同的性格所在。

先来认识一下Ruby，我的最爱之一。她偶尔会搞怪，却总是很妩媚；带有那么点神秘，却有着百分百的魅力。还记得英国保姆Mary Poppins[①]吗？她那个年代，保姆多半像C语言家族[②]的大多数语言那样，做什么都很利索，就是没什么人情味儿，而且枯燥死板、一成不变。其实，只要一勺糖[③]，一切都会不同。Mary Poppins从家务中寻找乐趣，以责任感唤起热情，做起家务来自然事半功倍。Ruby所做的也同样如此，但它用的不是食用糖，而是语法糖。作为Ruby的发明者，Matz并不担心编程语言的执行效率，而是把精力放在了提高程序员的编程效率上。

① *Mary Poppins*，DVD版，导演：Robert Stevenson（1964年）。发行商：加利福尼亚州洛杉矶市迪士尼影视公司（2004年）。（译者注：这部电影中译名为《欢乐满人间》，是迪士尼公司1964年根据同名小说拍摄的音乐电影。片中主人公Mary Poppins是一位法力高强的仙女保姆，她来到一户双亲工作忙碌、孩子生性顽皮的家庭，接替孩子们气走的前任保姆的工作。在她手中，枯燥的日常生活变得像魔幻世界一般精彩纷呈，繁重的家务劳动也具有了无穷无尽的乐趣，孩子们更是由此学会了以积极快乐的心态面对生活。Mary Poppins的行为感染了孩子们的父亲，让他明白除了工作、赚钱之外，值得珍惜的事物还有很多。最后，带着欣慰的心情，她离开了这个家庭。值得一提的是，饰演Mary Poppins这一角色的是著名英国影星Julie Andrews，这也是她的银幕处女作。一年后，她又饰演了《音乐之声》中的家庭教师——和Mary Poppins可谓是异曲同工、相映成趣。正是凭借这两个角色，她获得了1964年的奥斯卡最佳女演员奖以及1965年的奥斯卡最佳女演员提名。）

② C语言家族，指由C语言派生的众多编程语言，包括C、C++、C#、Java、Objective-C、D、Go等语言。可参见http://en.wikipedia.org/wiki/Category:C_programming_language_family。

③ 指的是Mary Poppins在影片中演唱的歌曲《一勺糖》。她借此歌告诉孩子们，虽然工作犹如苦药一般，但只要发现其中乐趣，就会像一勺糖压下苦药那般甘之如饴。

2.1 Ruby 简史

松本行弘（Yukihiro Matsumoto）大约在1993年发明了Ruby，大家多称他为Matz。从语言的角度看，Ruby出身于所谓的脚本语言家族，是一种解释型、面向对象、动态类型的语言。解释型，意味着Ruby代码由解释器而非编译器执行。动态类型，意味着类型在运行时而非编译时绑定。从这两方面看，Ruby采取的策略是在灵活性和运行时安全之间寻找平衡点，我们稍后还会深入讨论这一点。面向对象，意味着Ruby支持封装（把数据和行为一起打包）、类继承（用一棵类树来组织对象类型）、多态（对象可表现为多种形式）等特性。Ruby多年来一直默默蛰伏，只为等待一个恰当的出现时机。终于，随着Rails框架崭露头角，Ruby也在2006年前后一鸣惊人。在企业开发的丛林中跋涉了十年之后，Ruby指引人们重新找回了编程乐趣。尽管从执行速度上说，Ruby谈不上有多高效，但它却能让程序员的编程效率大幅提高。

松本行弘访谈录

我很高兴来到松本先生的家乡——日本松江市拜会松本先生。我们在谈话间聊到一些Ruby语言背后的设计思想，松本先生也解答了我向他提出的几个问题。

Bruce：你为什么要开发Ruby？

Matz：我从一开始摆弄计算机，就对编程语言产生了兴趣。编程语言不仅是用来编程的方法，还是思维的放大器，可以塑造思考编程的方式。所以很长一段时间，我都把编程语言当作一项兴趣爱好，下了不少功夫研究。我甚至实现了几门玩具语言，但都派不上什么用场。

1993年，当我看到Perl的时候，不知怎么的，这种混合了Lisp和Smalltalk特征的面向对象语言让我的灵感一下子迸发出来。我意识到Perl将成为一门可提高我们生产力的伟大语言。于是，出于自娱自乐的动机，我着手开发一门与之类似的语言，并将其命名为Ruby。刚开始的时候，开发Ruby还纯属业余爱好，处处都能按自己的口味设计。后来，世界各地的程序员开始渐渐接受这门语言及其背后的设计原则。它越来越受人们喜爱，这远远超出了我的预期。

Bruce：你最喜欢它哪一点呢？

Matz：我喜欢它寓编程于乐的方式。说到某个具体的技术点，我最喜欢的是“代码块”（block）。代码块即是一种易于控制的高阶函数，也为DSL（Domain-Specific Language，领域特定语言）及其他特性的实现提供了极大的灵活性。

Bruce：如果能让时光倒流，你想改变哪些特性？

Matz：我想去掉线程，加入actor（参与者）或一些更高级的并发特性。

无论你是否已对Ruby有所了解，都请一边阅读本章，一边留意Matz为设计这门语言所做的种种权衡。你可以看看他添加了哪些语法糖——那些打破了语言常规，不仅为程序员提供更加友好的体验，而且让代码更容易理解的小特性。还可以看看Matz在集合（collection）等处用到的代码块，体会一下它们如何发挥出梦幻般的效果。还有，尽可能去理解他在简单性和安全性之间、编码效率和程序性能之间所做的哪些让步和折中。

这就开始吧。先简单看看下面这几行Ruby代码，找找感觉：

```
>> properties = ['object oriented', 'duck typed', 'productive', 'fun']
=> ["object oriented", "duck typed", "productive", "fun"]
>> properties.each {|property| puts "Ruby is #{property}."}
Ruby is object oriented.
Ruby is duck typed.
Ruby is productive.
Ruby is fun.
=> ["object oriented", "duck typed", "productive", "fun"]
```

有了Ruby，我们又能带着笑容编程了。作为一门彻头彻尾的动态语言，它拥有令人惊叹的社区支持。它的各种实现版本都是开源的。它的社区里没有其他语言社区泛滥成灾的那种华而不实的框架，因为其商业支持大都来自于小公司。它虽在企业应用领域势头不显，但凭借编程效率上的优势，在Web开发等领域可谓如雷贯耳。

2.2 第一天：找个保姆

且不说Mary Poppins会什么魔法，她首先得是一名优秀保姆。当你刚上手学一门语言时，必须先了解如何用它去完成能用其他语言搞定的事。下面，我们即将开始和Ruby的首次接触。你可以把这当作一次彼此交流的机会。你们交流起来是否顺畅？有没有几分难以名状的尴尬？它有什么样的核心编程模型？采用何种方法处理类型？现在，让我们开始寻找答案吧。

2.2.1 快速起步

我承诺过，不会带你体验那种婆婆妈妈又老掉牙的安装过程。Ruby安装起来不过是小菜一碟。你只需移步http://www.ruby-lang.org/en/downloads/，找到你所用的平台，安装Ruby 1.8.6或更高版本即可。我在撰写本章时，用的是Ruby 1.8.7，而1.9版可能会有一些细微差别。如果你用Windows平台，可下载简便易用的一键安装包；如果你用OS X Leopard或更高版本的苹果平台，可在Xcode安装盘中找到Ruby。

输入`irb`可测试安装是否成功。如果没提示任何错误，你就放心阅读本章的剩余部分好了；如果提示错误，那也没什么好怕的，别人可能早就遇到过类似问题。只需把错误信息输入Google，解决方法可能就会在你面前出现。

2.2.2 从命令行执行Ruby

如果你尚未输入`irb`，现在马上输入。你会看到Ruby的交互命令行，其中可输入命令并获得反馈。输入下列命令：

```
>> puts 'hello, world'
hello, world
=> nil
>> language = 'Ruby'
=> "Ruby"
```

```
>> puts "hello, #{language}"
hello, Ruby
=> nil
>> language = 'my Ruby'
=> "my Ruby"
>> puts "hello, #{language}"
hello, my Ruby
=> nil
```


如果你对Ruby一无所知，这短短几行代码示例就暗藏了不少线索。第一，它告诉你，Ruby是解释执行的。确切地说，Ruby几乎总是解释执行的，但也有开发者正着手开发虚拟机，想把Ruby代码编译成字节码再执行；第二，我没在代码里声明任何变量①；第三，即使我没让Ruby返回任何值，这几行代码也都具有各自的返回值②。实际上，每条Ruby代码都会返回某个值。

此外，你还至少看见了两种形式的字符串。单引号包含的字符串表示它将不加改动地直接解释，双引号包含的字符串则会引发字符串替换。字符串替换是Ruby解释器所做的一种求值。在上面的示例中，Ruby把变量`language`的返回值替换进了字符串。好，继续前进。

2.2.3　Ruby的编程模型

当你新接触一门语言的时候，有些问题是需要首先去思考的，"这门语言的编程模型是什么"正是其中之一。这问题有时不那么好回答。你可能早就接触过C、Fortran、Pascal这类过程式语言。现如今，大部分人在用面向对象语言，不过它们大都带有过程式语言要素，比如，数字4在Java中就不是对象。你也许冲着函数式编程语言买的这本书。但某些函数式语言（如Scala）还加入了一些面向对象思想，可以说它们混合了多种编程模型。另外，也有很多其他编程模型。基于栈的语言（如PostScript或Forth），使用一个或多个栈作为该语言的核心特征。基于逻辑的语言（如Prolog），是以规则（rule）为中心建立起来的。原型语言（如Io、Lua和Self）用对象而不用类来作为定义对象甚至继承的基础。

Ruby是一门纯面向对象语言。在本章中你将看到，Ruby是如何深入挖掘面向对象思想的。先来看一些基本对象：

```
>> 4
=> 4
>> 4.class
=> Fixnum
>> 4 + 4
=> 8
>> 4.methods
=> ["inspect", "%", "<<", "singleton_method_added", "numerator", ...
 "*", "+", "to_i", "methods", ...
 ]
```

① 声明是指使用某变量之前先宣称该变量存在，或指出该变量的类型。对于Python、Ruby这样的语言来说，声明毫无意义，因为变量无需声明，即可直接初始化及赋值。

② 当然，`nil`也是值，因此它也是返回值。

虽然在代码最后的列表中，我省略了一些方法，但已经充分说明了问题。在Ruby中，一切皆为对象，就连每个单独的数字[①]也不例外。通过此例，我们可以看出，数字是`Fixnum`类型的对象，且调用`methods`方法可返回方法数组（Ruby用方括号表示数组）。实际上，我们可以用“.”符号调用对象具有的任意方法。

2.2.4　判断

程序编出来是用于判断决策的，因此在编程语言中，如何使用判断也就理所当然成为其核心思想，它影响我们用这门语言编码和思维的方式。Ruby的判断语句和其他大部分面向对象或过程式语言大同小异。看看下面的表达式：

```
>> x = 4
=> 4
>> x < 5
=> true
>> x <= 4
=> true
>> x > 4
=> false
>> false.class
=> FalseClass
>> true.class
=> TrueClass
```

也就是说，Ruby中有取值为`true`或`false`的表达式。和其他语言一样，Ruby中的`true`和`false`也是一等对象（first-class object）[②]。它们可用来执行下列涉及条件判断的代码：

```
>> x = 4
=> 4
>> puts 'This appears to be false.' unless x == 4
=> nil
>> puts 'This appears to be true.' if x == 4
This appears to be true.
=> nil
>> if x == 4
>>   puts 'This appears to be true.'
>> end
This appears to be true.
=> nil
>> unless x == 4
>>   puts 'This appears to be false.'
```

① 揣摩作者原意，当指不以集合形式（列表、元组等）出现的数字。

② 即与整数、浮点数等基本类型同等方式处理的对象。注意，这里说的对象并非特指面向对象语言中的对象，而是泛指编程语言中的类型。一等对象应具有以下几项性质：可存储于变量或数据结构中；可作为参数传递给函数；可作为返回值从函数返回；可在运行时创建。举例来说，C++中的对象就是一等对象，但其函数无法在运行时创建，所以不是一等对象；与之相反，函数式语言中的函数是一等对象，因为它既可以传递和返回，也可以在运行时动态创建。

```
>> else
>>   puts 'This appears to be true.'
>> end
This appears to be true.
=> nil
>> puts 'This appears to be true.' if not true
=> nil
>> puts 'This appears to be true.' if !true
=> nil
```


我非常喜欢Ruby的这种设计，它把条件句变得简单明了。当你使用`if`或`unless`时，既可选用块形式（`if` *`condition`*，*`statements`*，`end`），也可选用单行形式（*`statements`* `if` *`condition`*）。也许在某些人看来，`if`的单行形式令人作呕，但我却觉得，它仅仅用了一行代码，就清楚地表达了思想：

```
order.calculate_tax unless order.nil?
```

当然，你也可以用块形式写这行代码，但这样做会在本应单纯、连贯的思想中引入一些不必要的干扰。把简单的想法精炼到短短一行代码之中，别人读起来会轻松许多。此外，我还对`unless`颇为欣赏。你可以用`not`或`!`表达同样意图，但用`unless`表达要好得多。

`while`和`until`亦是如此：

```
>> x = x + 1 while x < 10
=> nil
>> x
=> 10
>> x = x - 1 until x == 1
=> nil
>> x
=> 1
>> while x < 10
>>   x = x + 1
>>   puts x
>> end
2
3
4
5
6
7
8
9
10
=> nil
```

注意，=用于赋值，而==用于判断是否相等。在Ruby中，每种对象都有自己特有的相等概念。数字对象的值相等时，它们相等。

你也可以用`true`和`false`之外的值作为表达式：

```
>> puts 'This appears to be true.' if 1
This appears to be true.
=> nil
>> puts 'This appears to be true.' if 'random string'
(irb):31: warning: string literal in condition
This appears to be true.
=> nil
>> puts 'This appears to be true.' if 0
This appears to be true.
=> nil
>> puts 'This appears to be true.' if true
This appears to be true.
=> nil
>> puts 'This appears to be true.' if false
=> nil
>> puts 'This appears to be true.' if nil
=> nil
```

也就是说，除了nil和false之外，其他值都代表true。C和C++程序员可得小心了，0也是true！

Ruby的逻辑运算符，跟C、C++、C#、Java差不多，但也稍有不同。and（也可写为&&）是逻辑与，or（也可写为||）是逻辑或。用这两种运算符验证表达式时，一旦表达式的值已能明确求出，解释器就不再继续执行后面的表达式代码[①]。如果想执行整个表达式的话，可以用&或|进行比较。下面，看看它们是如何运行的：

```
>> true and false
=> false
>> true or false
=> true
>> false && false
=> false

>> true && this_will_cause_an_error
NameError: undefined local variable or method 'this_will_cause_an_error'
    for main:Object
        from (irb):59
>> false && this_will_not_cause_an_error
=> false
>> true or this_will_not_cause_an_error
=> true
>> true || this_will_not_cause_an_error
=> true
>> true | this_will_cause_an_error
```

① 即逻辑表达式的短路求值。几乎所有语言都有此项特征。但不同的是，有些语言还有正常求值的逻辑运算符，即对所有子表达式一一求值。前面提到的4种语言中，C和C++只有短路求值，C#和Java虽二者皆有，但没有and和or。

```
NameError: undefined local variable or method `this_will_cause_an_error'
    for main:Object
        from (irb):2
        from :0
>> true | false
=> true
```

这段代码可谓一目了然，不必多加解释。我们一般会使用逻辑运算符的短路版本。

2.2.5 鸭子类型

接下来，我们学一点有关Ruby类型模型的内容。首先，你必须知道，当你错误使用类型时，Ruby能在多大程度上给予保护。这里说的就是类型安全（type safety）。强类型语言会对某些操作进行类型检查，并在其造成破坏前加以阻止。当你把代码提交给解释器或编译器，或是执行代码时，就会进行类型检查。看看下面的代码：

```
>> 4 + 'four'
TypeError: String can't be coerced into Fixnum
        from (irb):51:in `+'
        from (irb):51

>> 4.class
=> Fixnum
>> (4.0).class
=> Float

>> 4 + 4.0
=> 8.0
```

由此可见，Ruby是强类型语言[①]，这意味着发生类型冲突时，你将得到一个错误。另外，Ruby是在运行时而非编译时进行类型检查的。为了证明这一点，我打算比原计划提前一些介绍定义函数的方法。关键字def定义一个函数，但不会执行它。输入下列代码：

```
>> def add_them_up
>>   4 + 'four'
>> end
=> nil
>> add_them_up
TypeError: String can't be coerced into Fixnum
        from (irb):56:in `+'
        from (irb):56:in `add_them_up'
        from (irb):58
```

所以说，直到真正尝试执行代码时，Ruby才进行类型检查。这一概念称做动态类型。在采用

① 我对你撒了一点点谎，不过只有一点点。在后面两个例子中，你将看到，我在运行时改变了当前的类。从理论上说，用户能把类改得面目全非，并借此战胜类型保护，因此从最严格的角度来看，Ruby不是强类型语言。不过一般情况下，Ruby在大部分时间里都表现得像一门强类型语言。——原书注

静态类型系统的情况下，编译器和工具能捕获更多错误，因此Ruby不会像静态类型语言捕获的错误那么多。这是它的劣势。但Ruby的类型系统也有自己的潜在优势，即多个类不必继承自相同父类，就能以相同方式使用：

```
>> i = 0
=> 0
>> a = ['100', 100.0]
=> ['100', 100.0]
>> while i < 2
>>   puts a[i].to_i
>>   i = i + 1
>> end
100
100
```

刚才你看到的就是我们实际应用中常见的鸭子类型（duck typing）。这数组的第一个元素是`String`，第二个元素是`Float`，而把它们转换成整数的代码，却都用的是`to_i`。鸭子类型并不在乎其内在类型可能是什么。只要它像鸭子一样走路，像鸭子一样嘎嘎叫，那它就是只鸭子。在这个例子中，`to_i`就相当于嘎嘎叫。

对于面向对象设计的清晰性来说，鸭子类型至关重要。在面向对象设计思想中，有这样一个重要原则：对接口编码，不对实现编码①。如果利用鸭子类型，实现这一原则只需极少的额外工作，轻轻松松就能完成。举个例子，对象若有`push`和`pop`方法，它就能当作栈来用；反之若没有，就不能当作栈。

2.2.6　第一天我们学到了什么

学到这里，我们才好不容易把基础知识都过了一遍。Ruby是一门解释型语言。一切皆为对象，且易于获取对象的任何信息，如对象的各方法及所属类。它是鸭子类型的，且行为通常和强类型语言毫无二致，尽管一些学者会争论其中差别。它也是崇尚自由精神的语言，允许你做几乎一切事情，包括修改`NilClass`或`String`这样的核心类。现在，让我们放松一下，做一些自习。

2.2.7　第一天自习

你已经结束了和Ruby的首次约会，接下来该写写代码了。在这一阶段，你不必写出完整的程序，用`irb`执行Ruby语句就行。

找

- ❑ Ruby API文档。
- ❑ *Programming Ruby：The Pragmatic Programmer's Guide* [TFH08]的免费在线版本。
- ❑ 替换字符串某一部分的方法。
- ❑ 有关Ruby正则表达式的资料。

① 这是依赖反转原则（Dependency Inversion Principle，DIP）的一种实践应用。

- 有关Ruby区间（range）的资料。

做

- 打印字符串"Hello, world."。
- 在字符串"Hello, Ruby."中，找出"Ruby."所在下标。
- 打印你的名字十遍。
- 打印字符串"This is sentence number 1."，其中的数字1会一直变化到10。
- 从文件运行Ruby程序。
- 加分题：如果你感觉意犹未尽，还可以写一个选随机数的程序。该程序让玩家猜随机数是多少，并告诉玩家是猜大了还是猜小了。

（提示：rand(10)可产生0～9的随机数，gets可读取键盘输入的字符串，你要把输入字符串转换成整数。）

2.3　第二天：从天而降

想当年，Mary Poppins撑着伞、翩然降落于小镇的登场方式，绝对是那部电影最让人心驰目眩的一幕。要是搁现在，我的小孩才不会理解这样登场有什么好大惊小怪的。第二天中，你会亲身体验令Ruby大受欢迎的小魔法。你将学习对象、集合、类等基本构建单元的用法，还将学到代码块的基本要素。做好准备、睁大眼睛，见识一下这些小魔法吧。

2.3.1　定义函数

和Java、C#不同，你不必为了定义函数而把整个类都构建出来。用命令行就可以定义函数：

```
>> def tell_the_truth
>>   true
>> end
```

每个函数都会返回结果。如果你没有显式指定某个返回值，函数就将返回退出函数前最后处理的表达式的值。像所有其他事物一样，函数也是个对象。

我们稍后会讨论如何把函数作为参数传递给其他函数。

2.3.2　数组

数组是Ruby有序集合中的主力部队。虽说Ruby 1.9新引入了有序散列表，但总的来看，数组仍是Ruby最重要的有序集合。看看下面这段代码：

```
>> animals = ['lions', 'tigers', 'bears']
=> ["lions", "tigers", "bears"]
>> puts animals
lions
```

```
tigers
bears
=> nil
>> animals[0]
=> "lions"
>> animals[2]
=> "bears"
>> animals[10]
=> nil
>> animals[-1]
=> "bears"
>> animals[-2]
=> "tigers"
>> animals[0..1]
=> ['lions', 'tigers']
>> (0..1).class
=> Range
```

可以看到，Ruby的集合提供了一定的灵活性。如果访问任何未定义的数组元素，Ruby会直接返回nil。你也能发现一些不会让数组功能更强，但会让它更简便易用的特性。比如，animals[-1]返回倒数第一个元素，animals[-2]返回倒数第二个元素，以此类推。这样为了便于使用而添加的特性就是*语法糖*。表达式animals[0..1]看似有几分像语法糖，但其实不是。0..1是个Range（区间）对象，表示从0到1（包括0和1）的所有数字。

数组也可容纳其他类型的元素：

```
>> a[0] = 0
NameError: undefined local variable or method 'a' for main:Object
from (irb):23
>> a = []
=> []
```

啊，a现在还不是数组呢，我就把它当成数组来用了。这错误也提示我们，Ruby的数组和散列表的运行方式。实际上，[]是Array类的方法：

```
>> [1].class
=> Array
>> [1].methods.include?('[]')
=> true
>> #在ruby 1.9中，请使用[1].methods.include?(:[])
```

这样看来，[]和[]=不过是访问数组的语法糖而已。想正确使用它们，必须先在变量里放一个空数组，然后就能像下面这样操作变量：

```
>> a[0] = 'zero'
=> "zero"
>> a[1] = 1
=> 1
>> a[2] = ['two', 'things']
=> ["two", "things"]
```

```
>> a
=> ["zero", 1, ["two", "things"]]
```

从上面代码还可看出，数组元素不必具有相同类型。

```
>> a = [[1, 2, 3], [10, 20, 30], [40, 50, 60]]
=> [[1, 2, 3], [10, 20, 30], [40, 50, 60]]
>> a[0][0]
=> 1
>> a[1][2]
=> 30
```

多维数组也不过是数组的数组而已。

```
>> a = [1]
=> [1]
>> a.push(1)
=> [1, 1]
>> a = [1]
=> [1]
>> a.push(2)
=> [1, 2]
>> a.pop
=> 2
>> a.pop
=> 1
```

数组拥有极其丰富的API，可将其用作队列、链表、栈、集合，等等。现在，我们来看看Ruby的另一个主要集合——散列表。

2.3.3　散列表

记住，集合里面就是一个个用来存储对象的桶。在散列表的桶里，每个对象上都贴着一张标签。这标签就是键，而对象就是键所对应的值。散列表就是一串这样的键－值对：

```
>> numbers = {1 => 'one', 2 => 'two'}
=> {1=>"one", 2=>"two"}
>> numbers[1]
=> "one"
>> numbers[2]
=> "two"
>> stuff = {:array => [1, 2, 3], :string => 'Hi, mom!'}
=> {:array=>[1, 2, 3], :string=>"Hi, mom!"}
>> stuff[:string]
=> "Hi, mom!"
```

这段代码不算太复杂。散列表的运行机制很像数组，但不一定是整数下标，而是可以有任意类型的键。最后那个散列表很有趣，因为我在其中首次引入了符号（symbol）。符号是前面带有冒号的标识符，类似于`:symbol`的形式。它在给事物和概念命名时非常好用。尽管两个同值字符串在物理上不同，但相同的符号却是同一物理对象。我们可通过多次获取相同的符号对象标识符来证实这一点，像下面这样：

```
>> 'string'.object_id
=> 3092010
>> 'string'.object_id
=> 3089690
>> :string.object_id
=> 69618
>> :string.object_id
=> 69618
```

散列表有一些别出心裁的应用。比如，Ruby虽然不支持命名参数，但可以用散列表来模拟它。只要加进一颗小小的语法糖，你就能获得一些有趣的特性：

```
>> def tell_the_truth(options={})
>>   if options[:profession] == :lawyer
>>     'it could be believed that this is almost certainly not false.'
>>   else
>>     true
>>   end
>> end
=> nil
>> tell_the_truth
=> true
>> tell_the_truth :profession => :lawyer
=> "it could be believed that this is almost certainly not false."
```

该方法带一个可选参数。如果不传入该参数，options将设为空散列表，但如果传入:profession=>为:lawyer[①]，返回结果就有所不同。它不会返回true，但因为Ruby的求值机制将字符串也当作true处理，所以这和返回true几乎毫无差别。还需注意，这里的散列表不必用大括号括起来，因为将散列表用作函数的最后一个参数时，大括号可有可无。按理说，既然数组元素、散列表键、散列表值几乎可以任选类型，那么我们就能用Ruby构造出极为精妙的数据结构。然而，想真正做到这一点，还得先学会代码块才行。

2.3.4　代码块和yield

代码块是没有名字的函数。它可以作为参数传递给函数或方法，比如：

```
>> 3.times {puts 'hiya there, kiddo'}
hiya there, kiddo
hiya there, kiddo
hiya there, kiddo
```

大括号之间的代码就称作代码块。times是Fixnum类的方法，它会执行n次x，其中x是代码块，n是Fixnum对象的值。可以采用{/}或do/end两种界定代码块的形式，Ruby的一般惯例是：代码块只占一行时用大括号，代码块占多行时用do/end。代码块可带有一个或多个参数：

① 即传入一个包含键:profession和值:lawyer的散列表，如代码倒数第二行的:profession => :lawyer所示。这里profession为职业之意，lawyer为律师之意，因此该参数的语义是职业为律师。

```
>> animals = ['lions and ', 'tigers and', 'bears', 'oh my']
=> ["lions and ", "tigers and", "bears", "oh my"]
>> animals.each {|a| puts a}
lions and
tigers and
bears
oh my
```

上面这段代码能让你见识到代码块的威力，它指示Ruby在集合的每个元素上执行某些行为。仅仅用上少许代码块语句，Ruby就遍历了每一个元素，还把它们全都打印了出来。想亲手实践一下如何做到这点吗？看看以下自定义实现的`times`方法：

```
>> class Fixnum
>>   def my_times
>>     i = self
>>     while i > 0
>>       i = i - 1
>>       yield
>>     end
>>   end
>> end
=> nil
>> 3.my_times {puts 'mangy moose'}
mangy moose
mangy moose
mangy moose
```

这段代码打开一个现有的类，并向其中添加一个方法。此例中，添加的方法是`my_times`，它用`yield`调用代码块，并循环某个给定次数。代码块还可用作一等参数（first-class parameter）[①]，看看下面的例子：

```
>> def call_block(&block) ②
>>   block.call
>> end
=> nil
>> def pass_block(&block)
>>   call_block(&block)
>> end
=> nil
>> pass_block {puts 'Hello, block'}
Hello, block
```

这技术能让你把可执行代码派发给其他方法。在Ruby中，代码块不仅可用于循环，还可用于延迟执行：

```
execute_at_noon { puts 'Beep beep... time to get up'} ③
```

① 即可将代码块作为参数直接传递给函数（也就相当于将函数作为参数传递给函数），而无需包装在其他结构中传递。

② 在Ruby中，参数名之前加一个“&”，表示将代码块作为闭包传递给函数。

③ 代码块中的行为，也就是`puts 'Beep beep…time to get up'`并不会马上执行，而要等到调用该代码块相关的`yield`时才会执行。

执行某些条件行为：

```
……一段代码……
in_case_of_emergency do
  use_credit_card
  panic
end

def in_case_of_emergency
  yield if emergency?
end
……其他代码……
```

强制实施某种策略：

```
within_a_transaction do
  things_that
  must_happen_together
end
def within_a_transaction
  begin_transaction
  yield
  end_transaction
end
```

以及诸多其他用途。你会见到各种使用代码块的Ruby库，包括处理文件的每一行，执行HTTP事务中的任务，在集合上进行各种复杂操作等。Ruby简直就是个代码块的大联欢。

从文件中运行Ruby

随着代码示例越来越复杂，用交互命令行运行代码也越来越麻烦。命令行研究少量代码尚可，但多数情况下，还是把代码放入文件为好。创建一个名为hello.rb的文件，其中包含任意你想执行的Ruby代码，比如：

```
puts 'hello, world'
```

把文件保存到当前文件夹，然后从命令行执行以下命令：

```
batate$ ruby hello.rb
hello, world
```

集成开发环境（integrated development environment, IDE）虽然功能完善，但用它开发Ruby程序的人很少，大多数人还是乐于使用简便易用的文件编辑器。我最喜欢的编辑器是TextMate，包括vi、emacs在内的众多热门编辑器也都拥有Ruby插件。在你熟练掌握用文件运行Ruby程序之后，我们开始研究Ruby程序的可复用组件。

2.3.5　定义类

Ruby和Java、C#、C++一样，也有类和对象。想想饼干模板和饼干，类就是对象的模板。当然，Ruby也支持继承，但和C++不同，Ruby中的类只能继承自一个叫做*超类*的类。耳听为虚眼见为实，打开命令行，输入下列代码：

```
>> 4.class
=> Fixnum
>> 4.class.superclass
=> Integer
>> 4.class.superclass.superclass
=> Numeric
>> 4.class.superclass.superclass.superclass
=> Object
>> 4.class.superclass.superclass.superclass.superclass
=> nil
```

到目前为止，还没有什么难以理解的内容。对象是从类派生出来的。4的类是Fixnum，Fixnum继承自Integer，而Integer又继承自Numeric，最终，Numeric继承自Object。

看一下图2-1，它展示了这些事物是如何搭配在一起的。所有的事物，归根结底都继承自Object。一个Class同时也是一个Module。Class的实例将作为对象的模板。在我们的例子中，Fixnum是Class的一个实例，而4又是Fixnum的一个实例。每一个类同时也是一个对象。

```
>> 4.class.class
=> Class
>> 4.class.class.superclass
=> Module
>> 4.class.class.superclass.superclass
=> Object
```

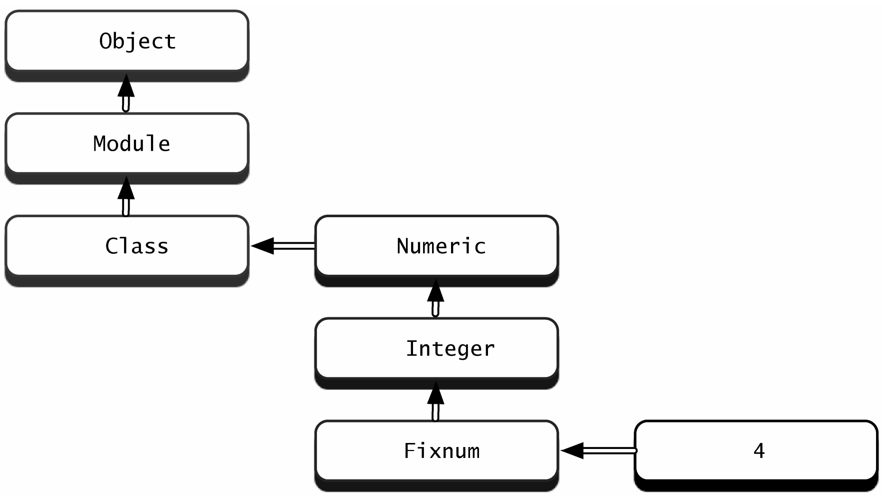


图2-1 Ruby元模型

如此看来，Fixnum派生自Class类。从这里开始，你可能会有些费解。Class继承自Module，Module又继承自Object，说到底，Ruby中的一切事物都有一个共同祖先——Object。

ruby/tree.rb

```
class Tree
```

```
  attr_accessor :children, :node_name

  def initialize(name, children=[])
    @children = children
    @node_name = name
  end

  def visit_all(&block)
    visit &block
    children.each {|c| c.visit_all &block}
  end

  def visit(&block)
    block.call self
  end
end

ruby_tree = Tree.new( "Ruby",
  [Tree.new("Reia"),
   Tree.new("MacRuby")] )

puts "Visiting a node"
ruby_tree.visit {|node| puts node.node_name}
puts

puts "visiting entire tree"
ruby_tree.visit_all {|node| puts node.node_name}
```

这个类用各种强大特性实现了一棵非常简单的树。它有三个方法：initialize、visit和visit_all，还有两个实例变量：children和node_name。initialize方法有特殊含义，类在初始化一个新对象的时候，会调用这个方法。

在这里，我有必要指出Ruby的一些惯例和规则。类应以大写字母开头，并且一般采用骆驼命名法，如CamelCase。实例变量（一个对象有一个值）前必须加上@，而类变量（一个类有一个值）前必须加上@@。实例变量和方法名以小写字母开头，并采用下划线命名法，如underscore_style。常量采用全大写形式，如ALL_CAPS。前面那段代码定义了一个树类。每棵树都有两个实例变量：@children和@node_name。用于逻辑测试的函数和方法一般要加上问号，如if test?。

attr关键字可用来定义实例变量。它有几种版本，其中最常用的版本是attr和attr_accessor。attr定义实例变量和访问变量的同名方法，而attr_accessor定义实例变量、访问方法和设置方法。[①]

前面那段程序真是塞进了不少东西，因此有些难以理解。它利用代码块和递归，使用户能访问树中所有节点。每个Tree类的实例都带有一个节点。initialize方法设置了children和

① 实际上，attr也可以定义设置方法，只需将true作为第二个参数传入即可，如attr_accessor :children, true。

node_name的初始值。visit方法调用传入的代码块。visit_all方法先对当前节点调用visit方法，然后对每个子节点递归调用visit_all方法。

类代码后面的剩余代码都用到了该类的API，先是定义一棵树，然后访问树中的一个节点，最后访问所有树节点。这产生了以下输出：

```
Visiting a node
Ruby

visiting entire tree
Ruby
Reia
MacRuby
```

类只是Ruby的诸多复杂概念之一。你在阅读图2-1上面的那段代码时，可能瞥见过模块（module）这个词。现在，让我们回过头去，仔细琢磨一下模块的概念。

2.3.6　编写Mixin

面向对象语言利用继承，将行为传播到相似的对象上。但对象若想继承并不相似的多种行为，一方面可通过允许从多个类继承（多继承）而实现，另一方面也可借助于其他解决方案。过往经验表明，多继承不仅复杂，且问题多多。Java采用接口解决这一问题，而Ruby采用的是模块。模块是函数和常量的集合。如果在类中包含了一个模块，那么该模块的行为和常量也会成为类的一部分。

通过下面这个类，我们可以把to_f方法添加到任意一个类上：

ruby/to_file.rb

```ruby
module ToFile
  def filename
    "object_#{self.object_id}.txt"
  end

  def to_f
    File.open(filename, 'w') {|f| f.write(to_s)}
  end
end
class Person
  include ToFile
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def to_s
    name
  end
```

```
end

Person.new('matz').to_f
```

代码一开始是模块定义。它定义了两个方法：to_f方法把to_s方法的输出写入文件，filename方法提供了写入文件的文件名。这里有件事很有趣：to_s在模块中使用，在类中实现，但定义模块的时候，实现它的类甚至还没有定义！这说明模块与包含它的类之间，是以一种相当隐秘的方式相互作用的。模块依赖的类方法通常不多。在Java中，这种依赖关系是显式的，即类会实现一个约束方法名的接口；而在Ruby中，这依赖关系是隐式的，即通过鸭子类型来实现。

对于Person类的细节，我们完全不感兴趣，这正是关键所在。的确，我们在Person类中包含了模块，但写入文件的能力，和这个类是不是Person没有一点儿关系。我们是通过混入（mix in）功能的方式，实现了在文件中添加内容的功能。我们可以对Person类添加新的mixin，也可以派生新的子类，这些子类虽然不了解mixin的具体实现，但仍然拥有mixin的功能。话说到这里，你应该已学会利用简明的单继承，先定义类的主要部分，然后用模块添加额外功能。这种由Flavors[①]引入，在上至Smalltalk[②]，下至Python的众多语言中采用的编程风格，就称作mixin。在这些语言中，带mixin的载体虽未必称作模块，但基本前提是一致的：使用单一继承结合mixin的方式，尽可能合理地把各种行为打包到一起。

2.3.7　模块、可枚举和集合

Ruby有两个至关重要的mixin：枚举（enumerable）和比较（comparable）。如果想让类可枚举，必须实现each方法；如果想让类可比较，必须实现<=>操作符。<=>被人们叫做太空船操作符，它比较a、b两操作数，b较大返回–1，a较大返回1，相等返回0。为避免方法实现之苦，集合已实现了许多便于使用的可枚举和可比较的方法。打开命令行输入以下代码：

```
>>  'begin' <=> 'end'
=> -1
>> 'same' <=> 'same'
=> 0
>> a = [5, 3, 4, 1]
=> [5, 3, 4, 1]
>> a.sort
=> [1, 3, 4, 5]
>> a.any? {|i| i > 6}
=> false
>> a.any? {|i| i > 4}
=> true
>> a.all? {|i| i > 4}
=> false
```

① Falvors是Lisp一种早期的面向对象扩展，由MIT人工智能实验室的Howard Cannon开发。

② Smalltalk是一门带有纯面向对象、动态类型、反射等特性的编程语言。它由施乐帕罗奥多研究中心（Xerox PARC）的Alan Kay、Dan Ingalls、Ted Kaehler等人于1969年开始开发，并于1972年面世。该语言对后来的许多编程语言及思想都有重大影响。

```
>> a.all? {|i| i > 0}
=> true
>> a.collect {|i| i * 2}
=> [10, 6, 8, 2]
>> a.select {|i| i % 2 == 0 } # even
=> [4]
>> a.select {|i| i % 2 == 1 } # odd
=> [5, 3, 1]
>> a.max
=> 5
>> a.member?(2)
=> false
```

只要集合中任一元素条件为真，any?就返回true；只有集合所有元素条件为真，all?才返回true。由于整数已通过Fixnum类实现了太空船操作符，因此可以调用sort方法排序，还可以调用min和max方法计算最小值和最大值。

我们也可以做一些基于集合（set）的操作。collect和map方法把函数应用到每个元素上，并返回结果数组。find方法找到一个符合条件的元素，而select和find_all方法均返回所有符合条件的元素。你还可以用inject方法计算列表的和与积：

```
>> a
=> [5, 3, 4, 1]
>> a.inject(0) {|sum, i| sum + i}
=> 13
>> a.inject {|sum, i| sum + i}
=> 13
>> a.inject {|product, i| product * i}
=> 60
```

inject方法看似复杂，实则不然。它后面跟一个代码块，里面有两个参数和一个表达式。inject会通过第二个参数，把每个列表元素传入代码块，这样代码块就能在每个列表项上执行操作。第一个参数是代码块上一次执行的结果。由于代码块第一次执行时，还没有上一次执行的结果，因此可以把初始值作为inject方法的参数传入。（如果不设初始值，inject会使用集合中的第一个值。）在添加了一些辅助代码之后，我们再来看看inject方法的执行过程：

```
>> a.inject(0) do |sum, i|
>>  puts "sum: #{sum}  i: #{i}   sum + i: #{sum + i}"
>>  sum + i
>>end
sum: 0  i: 5   sum + i: 5
sum: 5  i: 3   sum + i: 8
sum: 8  i: 4   sum + i: 12
sum: 12  i: 1   sum + i: 13
```

正如我们所料，上一行的结果总会显示为这一行的第一个值。有了inject方法，你能计算多个句子的单词总数，能找出段落各行的最长单词，还能做其他很多很多事情。

2.3.8　第二天我们学到了什么

这是你第一次见识到Ruby中的几颗糖和一点小魔法。你渐渐明白，Ruby是一门多么灵活的

语言。集合简直太好用了：用得最多的两个集合都带有众多API。Ruby关注的是程序员的效率，应用程序的效率是次要的。枚举模块让你品尝到Ruby良好设计的味道，单继承的面向对象结构虽不是什么新鲜事物，但Ruby的实现充满了符合直觉和实用的特性。如此程度的抽象并没有为Ruby带来本质上的变化，真正点石成金的魔法还在后面。

2.3.9　第二天自习

解答今天的问题会比昨天耗费更多精力。不过你用Ruby实践的时间也比昨天更长了，所以我想，你应该已经做好了准备。下面这些问题会迫使你更多地采用分析的思考方式。

找

- 分别找到用代码块和不用代码块读取文件的方法，用代码块有什么好处？
- 如何把散列表转换成数组？数组能转换成散列表吗？
- 你能循环遍历散列表吗？
- Ruby的数组能当作栈来用，它还能用作哪些常用的数据结构？

做

- 有一个数组，包含16个数字。仅用`each`方法打印数组中的内容，一次打印4个数字。然后，用可枚举模块的`each_slice`方法重做一遍。
- 我们前面实现了一个有趣的树类`Tree`，但它不具有简洁的用户接口，来设置一棵新树，为它写一个初始化方法，接受散列表和数组嵌套的结构。写好之后，你可以这样设置新树：`{'grandpa' => { 'dad' => {'child 1' => {}, 'child 2' => {} }, 'uncle' => {'child 3' => {}, 'child 4' => {} } } }`。
- 写一个简单的grep程序，把文件中出现某词组的行全都打印出来。这需要使用简单的正则表达式匹配，并从文件中读取各行。（这在Ruby中超乎想象地简单。）如果你愿意的话，还可以加上行号。

2.4　第三天：重大改变

Mary Poppins之所以成功，秘诀在于她不仅把家务变得妙趣横生，还让人们在面对繁重家务时，学会调动热情、发挥想象力，从而使家务劳动轻松许多。你可能不会像Mary Poppins那样，把Ruby彻底改造一番，而是选择稳妥行事，只用它去做一些其他语言已可从容完成的任务。然而，只有改变一门语言的本来面目和行为方式，你才算真正掌握了赋予编程无穷乐趣的魔法。在本书的每一章，你都会看到一个有价值的问题，而这问题，正是那一章的语言所擅长解决的。对Ruby而言，这问题是元编程（metaprogramming）。

元编程，说白了就是“写能写程序的程序”。Rails核心的ActiveRecord框架，就用元编程实现了一门简便易用的语言，以便编写连接数据库表的类。如果给`department`（部门）写个`ActiveRecord`类，写出来可能像下面这样：

```ruby
class Department < ActiveRecord::Base
  has_many :employees
  has_one :manager
end
```

has_many和has_one是两个Ruby方法，它们会把建立一对多关系将要用到的所有实例变量和方法都添加进来。这个类的定义看着就像个英文句子一样，丝毫没有其他数据库框架中常见的干扰和负担。下面，我们来看几种可用在元编程当中的技术。

2.4.1 开放类

你和“开放类”曾有过一面之交[1]，它可以随时改变任何类的定义，常用于给类添加行为。这里有个非常好的例子，来自Rails框架，它为NilClass添加了一个方法：

ruby/blank.rb

```ruby
class NilClass
  def blank?
    true
  end
end

class String
  def blank?
    self.size == 0
  end
end

["", "person", nil].each do |element|
  puts element unless element.blank?
end
```

在某个类名上首次调用class关键字会定义一个类，但如果该类已定义过，再调用class会修改先前的类定义。以上代码对两个现有的类——NilClass和String添加了一个blank?方法。添加这方法是因为我检查字符串状态时，经常想看看字符串是否为空，而字符串既可能带一个值，也可能是空字符串，还可能是nil。有了这个小巧的惯用法，仅需一次方法调用，就能快速检出后两种空状态，因为blank?都会返回true。不管字符串用的是哪个类，只要提供blank?方法，这么检查就没问题。只要走起来像鸭子、叫起来也像鸭子，它就是只鸭子，没必要去做什么抽血化验。

看看这段代码到底做了些什么。你想要一把削铁如泥的匕首，而Ruby正是将这样一把匕首递到你手中，于是，你得以开放String和NilClass两个类，对它们进行了重定义。利用重定义，我们甚至能让Ruby完全瘫痪，比如重定义Class.new方法。对于开放类来说，这里的权衡主要考虑了自由。有这种随时重定义任何类或对象的自由，我们就能写出极为通俗易懂的代码。不过你也要明白，自由越大、能力越强，担负的责任也越重。

① 参见第2.3.4节“代码块和Yield”。

实现用于领域编程的特定语言时，开放类特别有用，其中一种常用情况，是通过语言表示业务领域的度量单位，例如，考虑下面这个以英寸为距离单位的API：

ruby/units.rb

```
class Numeric
  def inches
    self
  end

  def feet
    self * 12.inches
  end

  def yards
    self * 3.feet
  end

  def miles
    self * 5280.feet
  end

  def back
    self * -1
  end

  def forward
    self
  end
end

puts 10.miles.back
puts 2.feet.forward
```

使用开放类就可以像上面那样采用最简单的语法轻松实现用英寸表示的距离。不过，除了开放类，还有别的技术能让Ruby的威力更加强大。

2.4.2　使用method_missing

Ruby找不到某个方法时，会调用一个特殊的调试方法显示诊断信息。该特性不仅让Ruby更易于调试，有时还能实现一些不易想到的有趣行为。只需要覆盖method_missing方法，我们就可以实现这些行为。思考一下，如何编写一个表示罗马数字的API。或许你觉得可以用方法调用轻松实现这个API，类似Roman.number_for "ii"。说实话，这样做也不坏，毕竟没有括号、分号什么的捣乱，不过用Ruby，我们能做得更漂亮：

ruby/roman.rb

```
class Roman
  def self.method_missing name, *args
    roman = name.to_s
    roman.gsub!("IV", "IIII")
    roman.gsub!("IX", "VIIII")
    roman.gsub!("XL", "XXXX")
    roman.gsub!("XC", "LXXXX")

    (roman.count("I") +
     roman.count("V") * 5 +
     roman.count("X") * 10 +
     roman.count("L") * 50 +
     roman.count("C") * 100)
  end
end

puts Roman.X
puts Roman.XC
puts Roman.XII
puts Roman.X
```

这段代码十分简明，是将`method_missing`方法用于实践的极佳示例。我们先是覆盖了`method_missing`方法。通过该方法的参数列表，可获得未找到方法的名称和参数，这个例子中，我们只对名称感兴趣。首先，我们把名称转换为字符串；然后，`iv`和`ix`这样的特殊数字形式[①]，将被替换为更容易计数的字符串；最后，是对罗马数字进行计数，并将计数结果与数字的值相乘。这个API比前面提到的那个简单得多，对比一下`Roman.i`和`Roman.number_for "i"`就能看出来。

然而，这样做也要付出代价：我们写的类调试起来会比过去困难得多，因为Ruby再也不会告诉你找不到某个方法！我们当然想严格检查错误，确保方法接受了正确的罗马数字。但如果不熟悉`method_missing`方法的上述用法，想找到`Roman`类如何实现`ii`方法都会很困难，更别说检查错误了。尽管如此，`method_missing`方法仍然是你武器库中的一把利器。只是在用它的时候一定要三思而行。

2.4.3 模块

说到Ruby最流行的元编程方式，非模块莫属。仅在模块中写上寥寥数行代码，就可以实现`def`或`attr_accessor`关键字的功能。你还可以通过一些令人惊叹的方式扩展类定义，其中一种技术是设计自己的DSL（domain-specific language，领域特定语言），再用DSL定义自己的类[②]。该

① 罗马数字一共有七个：I、V、X、L、C、D、M，分别代表：1、5、10、50、100、500、1000，具体规则请参见维基百科的“罗马数字”词条。文中程序只考虑到C，也就是100。该程序为计算方便，把数字的“左减”形式一律转换成“右加”形式（尽管不符合规则），再对转换后的数字进行计算。

② DSL可以为某个特定领域量身打造一门语言。Ruby中就有个广为人知的例子：`ActiveRecord`的持久化框架使用DSL，把一个类映射到一张数据库表上。——原书注

DSL在模块中定义各种方法，这些方法添加了对类进行管理所必需的全部方法和常量。

首先，用常见的超类（superclass）剖析程序示例。从下面的程序中，你将会了解到我们稍后用元编程编写的将是什么样的类。这程序很简单——根据类名，打开相应的CSV文件：

ruby/acts_as_csv_class.rb

```
class ActsAsCsv
  def read
    file = File.new(self.class.to_s.downcase + '.txt')
    @headers = file.gets.chomp.split(', ')

    file.each do |row|
      @result << row.chomp.split(', ')
    end
  end

  def headers
    @headers
  end

  def csv_contents
    @result
  end

  def initialize
    @result = []
    read
  end
end

class RubyCsv < ActsAsCsv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

这个基础类定义了4个方法：`headers`和`csv_contents`是两个仅返回实例变量值的访问器，`initialize`初始化读取结果，而`read`承担了这个类的大部分工作——打开文件，读取表头，把表头切分成一个个字段，再循环各行，把每一行的内容放入数组。这个读取CSV文件的类，因为没处理引号之类的特殊情况[①]，所以功能实现并不完整，但其思路不难理解。

下面的代码还是读取CSV文件，但这回，用一个叫做宏（macro）的模块方法添加类行为。宏经常根据环境变化改变类行为。在示例中，宏开放类，并把所有与CSV文件相关的行为复制到类中。

① CSV文件有一些特殊规则，比如，由于逗号和换行符用来切分字段和行，而且CSV文件忽略字段内容前后的空格，因此若内容本身带有逗号、换行符或前后空格时，必须用双引号把内容括起来，而若内容本身带有双引号，还必须用两个双引号代替单个双引号。这些都需要在读取时额外处理。

ruby/acts_as_csv.rb

```
class ActsAsCsv
  def self.acts_as_csv

    define_method 'read' do
      file = File.new(self.class.to_s.downcase + '.txt')
      @headers = file.gets.chomp.split(', ')

      file.each do |row|
        @result << row.chomp.split(', ')
      end
    end

    define_method "headers" do
      @headers
    end

    define_method "csv_contents" do
      @result
    end

    define_method''initialize''do
      @result = []
      read
    end
  end
end

class RubyCsv < ActsAsCsv
  acts_as_csv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

元编程发生在acts_as_csv宏当中，它对我们想添加到目标类上的所有方法都调用了define_method。现在，当目标类调用acts_as_csv时，宏代码会为目标类定义4个方法。

所以说，acts_as_csv宏不过是添加了一些方法，而这些方法本可以通过继承轻松添加。这样设计看来改进不大，但好戏还在后面。我们看看在模块中，同样的行为是如何实现的：

ruby/acts_as_csv_module.rb

```
module ActsAsCsv

  def self.included(base)
    base.extend ClassMethods
  end
```

```
    module ClassMethods
      def acts_as_csv
        include InstanceMethods
      end
    end

    module InstanceMethods

      def read
        @csv_contents = []
        filename = self.class.to_s.downcase + '.txt'
        file = File.new(filename)
        @headers = file.gets.chomp.split(', ')

        file.each do |row|
          @csv_contents << row.chomp.split(', ')
        end
      end

      attr_accessor :headers, :csv_contents

      def initialize
        read
      end

    end

  end

  class RubyCsv  # 没有继承，你可以自由添加
    include ActsAsCsv
    acts_as_csv
  end

  m = RubyCsv.new
  puts m.headers.inspect
  puts m.csv_contents.inspect
```

只要某个模块被另一模块包含，Ruby就会调用该模块的included方法。记住，类也是模块。在ActsAsCsv模块的included方法中，我们扩展了名为base的目标类（即RubyCsv类）。该模块还为RubyCsv类添加了类方法。其中，acts_as_csv是唯一的类方法。接下来，acts_as_csv方法又打开了RubyCsv类，并在类中包含了所有实例方法。如此这般，我们就写了一个会写程序的程序。

所有这些元编程技术的有趣之处在于，程序可以根据它应用时的状态而改变。ActiveRecord利用元编程，动态添加与数据库中的列有相同名称的访问器。有些XML框架如builder，可允许用户通过method_missing方法定义自定义标签，以提供更加美观的语法。当代码的语法变得更美观，阅读代码的读者也就不必在语法问题上大伤脑筋，从而能更好地理解代码本身表达的意图。这正是Ruby的威力所在。

2.4.4 第三天我们学到了什么

在本节中，你学会用Ruby定义自己的语法，以及动态地改变类，这两种编程技术都可以归到元编程当中。你写的每一行代码都会有两类读者，一类是电脑，一类是人。有时候，想创建既便于编译器或解释器加工又利于人类理解的代码并不容易。借助元编程的威力，你可以做到尽量缩短正确的Ruby语法与日常用语之间的距离。

一些最出色的Ruby框架，如`Builder`和`ActiveRecord`，都会为了改善可读性而特别依赖元编程。在本节中，利用开放类技术编写了一个鸭子类型接口，为`String`对象和`nil`提供了`blank?`方法，从而大大减少了用其他语言完成类似任务可能出现的大量杂乱无章的代码。你还见到了一段多次开放类的代码。利用`method_missing`方法写出了漂亮的罗马数字代码。最后，采用模块定义了DSL，又用它解析了CSV文件。

2.4.5 第三天自习

做

修改前面的CSV应用程序，使它可以用`each`方法返回`CsvRow`对象。然后，在`CsvRow`对象上，对某个给定标题，用`method_missing`方法返回标题所在列的值。

比如，对于包含以下内容的文件：

```
one, two
lions, tigers
```

API可以像下面这样操作：

```
csv = RubyCsv.new
csv.each {|row| puts row.one}
```

这会打印出`"lions"`。

2.5 趁热打铁

在本章中，我们讨论了不少内容。我希望你能明白为什么将Ruby比作Mary Poppins。在几十个Ruby会议上做过发言后，很多人都对我表示，他们之所以热爱Ruby，正是因为它能带来无穷的乐趣。对一个C家族语言（包括C++、C#、Java等）日渐泛滥的行业来说，Ruby犹如徐徐吹来的一阵清风，为人们带来了些许清爽的感觉。

2.5.1 核心优势

Ruby的纯面向对象可以让你用一致的方式来处理对象。鸭子类型根据对象可提供的方法，而不是对象的继承层次，实现了更切合实际的多态设计。Ruby的模块和开放类，使程序员能把行为紧密结合到语法上，这大大超越了类中定义的传统方法和实例变量。

Ruby作为脚本语言或带有合理扩展需求的Web开发语言而言，可以说是相当理想的。它的生产力很强，但其中某些提高生产力的特性使得Ruby难以编译，在性能上也有所损失。

1. 脚本

Ruby是一门梦幻般的脚本语言。Ruby可以非常出色地完成许多任务，例如编写胶水代码（glue code）合并两个互不兼容的应用程序，编写爬虫程序抓取股票报价或图书价格的网页，运行本地编译环境或自动测试等。

作为一门大部分主流操作系统都可使用的语言，Ruby是一个不错的脚本环境语言。在基本库之外，Ruby还包含了各种各样的库，以及数以千计的gem[①]或预打包插件，它们可用在加载CSV文件、处理XML、操作底层互联网API等任务当中。

2. Web开发

Rails现已成为有史以来最成功的Web开发框架之一，其设计理念以广为人知的模型—视图—控制器（model-view-controller）范型为基础。数据库元素和应用程序元素都有很多命名规范，因此不必进行任何配置就可以构建出典型的Rails应用程序，而且，该框架还有不少用来处理棘手的生产问题的插件：

- Rails应用程序的架构总是保持一致，并且早已为人所熟知；
- Migration处理数据库schema中的变化；
- 几个可减少配置代码数量的规范都有丰富文档；
- 有众多不同功能的插件可供选择。

3. 市场投放时间

说到Ruby和Rails的成功，我认为生产力是非常重要的因素。2005年前后，随便在旧金山扔块石头，肯定会砸到一个为创业公司工作，而且采用Rails开发的家伙。甚至现如今，Ruby在这些公司中依旧硕果累累，这其中就包括我自己的公司。优美的语法加上编程社区、工具、各种插件，这些因素组合在一起，使Ruby拥有极其巨大的威力。利用各种各样的Ruby gem，你可以找到某位冲浪者的邮政编码，也可以获得方圆80公里内所有邮政编码。[②]你可以处理图像和信用卡，可以利用Web服务完成任务，还可以在多种编程语言间通信。

很多大型商业网站都使用了Ruby和Ruby on Rails。Twitter最开始就是用Ruby实现的。借助Ruby无比强大的生产力，它迅速发展为一家规模庞大的网站。之后，他们用Scala重写了Twitter的核心。这告诉我们两件事：第一，对于快速开发一个可推向市场的合格产品，Ruby是一门非常好用的语言；第二，从某种程度上说，Ruby在可扩展性上有所局限。

在采用分布式事务、容错消息传递（fail-safe messaging）和国际化等机制的正规大型企业中，Ruby的作用常被人低估，但Ruby有能力做好上述所有事情。很多时候，我们应根据实际情况，采用适当的应用框架和可扩展性，然而现在，大部分人关注的是足以打造下一个eBay的可扩展性，

① 打包的Ruby应用程序或库称作gem，具有名称和版本号。在本地可以用gem命令管理gem包，这包括安装、卸载、查询等。

② 这种可提供地理信息的gem很多，比如Geokit。

却到交付软件的那一天什么都拿不出来。大多数情况下，Ruby都会充分考虑到市场投放时间的压力，而这种压力正是很多企业要面对的。

2.5.2　不足之处

再好的语言也不会对所有应用都表现完美。Ruby也有其局限性。下面，我们看看其中最主要的一些局限。

1. 性能

Ruby的最大弱点就是性能。没错，Ruby是越来越快了。在某些应用场景中，1.9版甚至比过去快了10倍之多。Evan Phoenix写的新虚拟机Rubinius初步实现了用即时编译器（just-in-time compiler）编译Ruby代码。用这种方法观察大段代码在企业中应用的模式，以判断哪些代码可能多次使用。这对于Ruby的运行方式来说相当合适，因为Ruby语法提供的线索通常不足以完成编译。别忘了，类的定义任何时间都可能改变。

Matz也十分清楚，他写Ruby是为了改善程序员的体验，而不是优化语言的性能。Ruby正是凭借它的许多特性（比如开放类、鸭子类型、`method_missing`等），击败了可编译并由此提升性能的语言。

2. 并发和面向对象编程

面向对象编程有一个重大弱点：该编程模型成立的一切前提条件，都建立在一种思想（围绕状态包装一系列行为）的基础之上，但通常，状态是会发生改变的。于是，程序中存在并发时，这种编程策略就会引发严重问题。最好的情况下，Ruby会产生大量的资源竞争；最坏的情况下，面向对象语言几乎无法在并发环境下调试程序，也无法可靠地测试程序。在我写这本书的时候，Rails团队刚开始解决如何有效地管理并发的问题。

3. 类型安全

我是鸭子类型的坚定支持者。在这种类型策略下，你通常可用简洁而清晰的代码对事务进行完美地抽象。但使用鸭子类型也是有代价的。静态类型可提供一整套工具，可以更轻松地构造语法树，也因此能实现各种IDE（集成开发环境）。对Ruby来说，实现IDE就困难得多，而且直到现在，也没几个人真正用IDE开发Ruby程序。[①]我多次从心底哀叹Ruby没有一个IDE形式的调试器。我想，有类似体验的人不会只有我一个。

2.5.3　最后思考

综上所述，Ruby的核心优势是它的语法和灵活性，根本的不足之处大概是性能，尽管应用于很多场景时，它的性能都还过得去。总之，在面向对象开发中，Ruby是一门优秀的语言。对于合适的应用，Ruby驾轻就熟。像其他工具一样，只要用它解决一组合适的问题，你就几乎不会失望。还有，使用它的过程中最好别闭眼，不然你就会错过一个个精彩的小魔法。

① 这里的表述信息有些过时了，现在有很多非常好的IDE（如Jet Brain的Ruby Mine），Eclipse和Net beans也都有对应的插件。

第 3 章 Io

问题不是“我们要干点儿什么”而是“我们有什么不能干”。

——Ferris Bueller

先来认识一下Io。和Ruby一样，Io懂得变通，行事不拘小节。他血气方刚、聪明过人，想了解他不难，想猜透他要做什么可就难了，活脱一个Ferris Bueller①。如果你愿意享受喧嚣热闹的狂欢，跟着Io逛逛绝对没错，他什么都会带你尝试一遍。和他在一起，你可能会有最美妙刺激的体验，但你老爸的车也可能变成一堆废铜烂铁。不过，无论发生什么，你都决不会无聊。正如本页最上方Ferris所说，没那么多清规戒律束手束脚。

3.1 Io 简介

2002年，Steve Dekorte发明了Io语言，这名字要写成大写的I后接一个小写的o。如同Lua、JavaScript一样，Io是一种原型语言，这意味着每个对象都是另一个对象的复制品。

Steve一开始只想写个程序练练手，以便弄清楚解释器的工作方式。没想到Io问世之后，竟成为业余爱好者们推崇的语言，并且发展到今天，依旧保持着小巧的特点。一刻钟学会语法，半小时学会基本原理，这些都不在话下。然而，到了学习Io库的阶段，花的时间就要多一些了，因为这门语言的复杂性和丰富性，统统来自于库的设计。

如今的大多数Io社区，都致力于把Io作为带有微型虚拟机和丰富并发特性的可嵌入语言来推广。Io的核心优势是拥有大量可定制语法和函数，以及强有力的并发模型。它的简单语法和原型编程模型都值得我们重点关注。我发现：在了解Io之后，我对JavaScript运行机制的理解也变得透彻许多。

① *Ferris Bueller's Day Off*，DVD版，导演：John Hughes（1986年）。发行商：加利福尼亚州好莱坞派拉蒙影业公司（1999年）。（译者注：这部电影中译名为《春天不是读书天》，是一部轻松诙谐、带有反教育色彩的喜剧片。它讲述一位十七岁高中生Ferris Bueller，同女友及好友一起逃学，驾着好友父亲珍藏的法拉利GT 250跑车，前往附近的芝加哥市畅快淋漓地游玩了一整天，晚上又准时回家、瞒过父母的故事。该影片处处传达了一种自由精神，尤其影片结尾主角所说的“世界如此精彩，如果不懂得驻足欣赏，那就会错过很多风景”更是点出该片主旨所在。）

3.2　第一天：逃学吧，轻松一下

初见Io与初见其他语言并无不同，你也要花上点时间不断敲打键盘，彼此之间才会渐渐熟悉。当然，如果你们没有在上历史课前，站在教室旁边的走廊上，进行一番令人窒息的交谈①，相互了解起来会容易许多。这就和Io一起逃学，去做些真正激动人心的事情吧！

名字有时带有欺骗性，但从Io这个名字中，我们还是能看出不少东西。一方面，这个名字起得有欠考虑（试过用Google搜索“Io”吗？）②；另一方面，这名字也处处闪耀着智慧光芒。它只有两个字母，还都是元音。而Io语法也正如其名，既简单又直接。Io语法只不过是把消息全部串联起来，每条消息都会返回一个对象，每条消息也都带有置于括号内的可选参数。在Io中，万事万物皆为消息，且每条消息都会返回另一接收消息的对象。Io这门语言没有关键字，有的只是少量在行为上接近于关键字的字符。

3

用Io的时候，你不必既操心类又操心对象。你只需和对象打交道，必要时把对象复制一下就行。这些被复制的对象就叫做*原型*。Io是我们介绍的第一门、也是仅有的一门基于原型的语言。在原型语言中，每个对象都不是类的复制品，而是一个实实在在的对象。此外，Io还能带你无限接近面向对象的Lisp。现在就对Io能否持续发挥影响力下断言尚为时过早，但简明的语法无疑是其成为利器的巨大优势。你将在第三天看到Io构思精妙的并发库，以及强大而优雅的消息语义。反射在Io当中也是无所不在。

3.2.1　开场白

下面请打开解释器，开始这场狂欢。在http://iolanguage.com可找到Io解释器，下载并安装它。键入`io`可打开解释器，然后输入经典的“Hello, World”程序：

```
Io> "Hi ho, Io" print
Hi ho, Io==> Hi ho, Io
```

你完全能看出这段代码是怎么做到这点的。你发送了`print`消息给字符串`"Hi ho, Io"`，接收者在左边，消息在右边。这里没有任何语法糖，你不过是把消息发送给对象而已。

在Ruby中，你可对某个类调用new创建一个新对象。通过定义类，可以创建一个新的对象种类。而Io不区分类和对象。你可通过复制现有对象创建新对象。现有对象就是原型：

```
batate$ io
Io 20090105
Io> Vehicle := Object clone
==>  Vehicle_0x1003b61f8:
  type             = "Vehicle"
```

`Object`是根对象。我们发送`clone`消息过去，它会返回一个新对象。我们把这个返回的新对象赋值给`Vehicle`。这里的`Vehicle`不是类，也不是用来创建对象的模板，它只是个对象，是一

① 此处是借《春天不是读书天》之中的一幕场景作为比喻。
② 你可以再试试用Google搜索“Io language”。——原书注

个基于Object原型的对象。我们可以继续用Vehicle做一些交互：

```
Io> Vehicle description := "Something to take you places"
==> Something to take you places
```

对象还带有槽。你可以把一组槽想象成散列表，通过键就能引用到任何一个槽。你既可以用:=给槽赋值，这种情况下，当槽尚不存在时，Io会创建出一个槽；也可以用=给槽赋值，这种情况下，当槽不存在时，Io会抛出一个异常。我们刚才创建了一个名为description的槽。

```
Io> Vehicle description = "Something to take you far away"
==> Something to take you far away
Io> Vehicle nonexistingSlot = "This won't work."

  Exception: Slot nonexistingSlot not found.
    Must define slot using := operator before updating.
  ---------
  message 'updateSlot' in 'Command Line' on line 1
```

向Vehicle对象发送槽的名字，可以获取槽中的值：

```
Io> Vehicle description
==> Something to take you far away
```

其实，对象在概念上要比一组槽更丰富一些。我们可以像这样看到Vehicle上所有槽的名字：

```
Io> Vehicle slotNames
==> list("type", "description")
```

我们向Vehicle对象发送了slotNames方法，并返回了一个槽名列表。可以看到，Vehicle对象有两个槽。你已经知道有description槽，但还有一个槽叫做type。任何对象都有type这个槽：

```
Io> Vehicle type
==> Vehicle
Io> Object type
==> Object
```

再过几小节，我们就会介绍类型（type）。现在，我们暂且认为type槽代表你当前处理对象的种类。不过一定要记住，该类型是对象而不是类。下面是迄今为止已学到的内容：

- 通过复制其他对象来制造对象；
- 对象是一组槽；
- 通过发送消息来获取槽的值。

你已经见识到了Io的简单有趣之处，学会它们毫不困难。然而，这些仅仅是Io的皮毛。接下来，我们将学习继承。

3.2.2　对象、原型和继承

在本节中，我们要和继承打交道。话说一辆小汽车（car），它当然也是一种交通工具（vehicle）。想想看，我们该如何构造一个法拉利赛车（ferrari）对象，而且它还是小汽车的实例？如果是面向对象语言，你可以像图3-1那样设计。

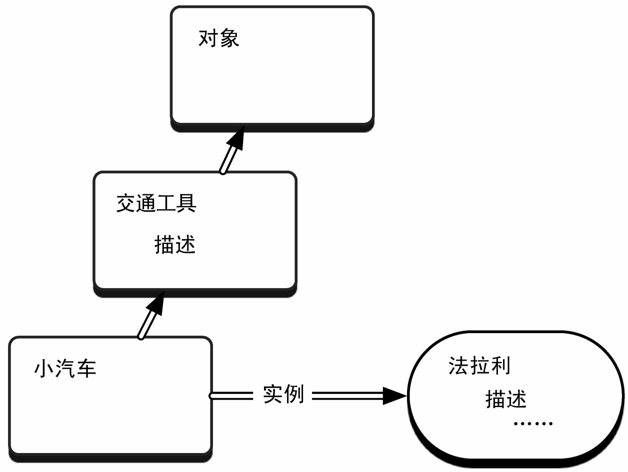


图3-1　某种面向对象设计

那么，在原型语言中，我们该如何解决这问题？除了在上一小节中创建的那些对象之外，我们还需要另外一些对象才行。首先，创建下面这个对象：

```
Io> Car := Vehicle clone
==>  Car_0x100473938:
  type             = "Car"

Io> Car slotNames
==> list("type")
Io> Car type
==> Car
```

在Io语言中，我们通过把`clone`消息发送给`Vehicle`原型，创建一个名为`Car`的新对象。接着，我们把`description`发送给`Car`：

```
Io> Car description
==> Something to take you far away
```

`Car`没有`description`槽，因此Io会把`description`消息转发给`Car`的原型`Vehicle`，并在`Vehicle`中找到这个槽。这机制十分简单，但却非常强大。然后，我们再来创建另一辆小汽车。不过这次，我们把它赋值给`ferrari`：

```
Io> ferrari := Car clone
==>  Car_0x1004f43d0:

Io> ferrari slotNames
==> list()
```

嘿，这下连type槽都没了。这是因为，依照Io的惯例，其类型应以大写字母开头。如果现在调用type槽，会得到它原型的类型：

```
Io> ferrari type
==> Car
```

这就是Io对象模型的工作方式。对象不过是槽的容器而已。发送槽名给对象可获得该槽。如果该槽不存在，则调用父对象的槽。你要理解的全部内容就是这些。这里没有类，也没有元类。你手里也不会有接口或模块，有的只是对象，如图3-2所示。

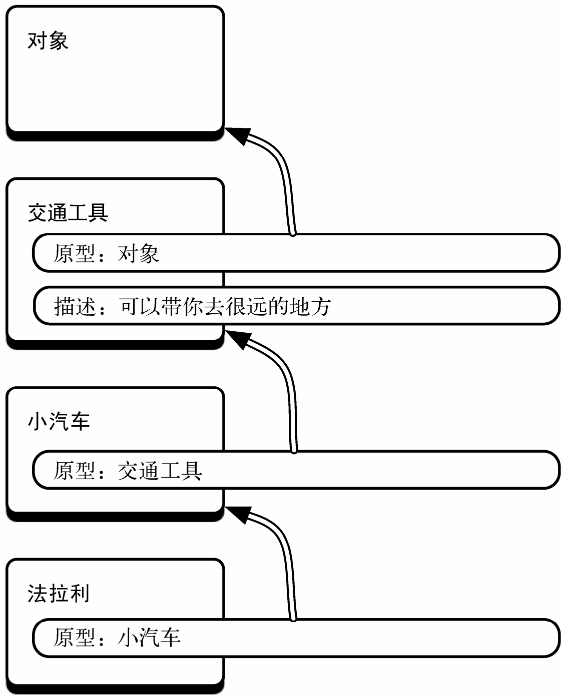


图3-2　Io中的继承

Io的类型是一种非常好用的机制。从惯用法的角度说，以大写字母开头的对象是类型，因此Io会对它设置`type`槽。而类型的复制品若以小写字母开头，则会调用它父对象的`type`槽。类型仅仅是帮助Io程序员更好地组织代码的工具。

如果想让法拉利赛车也成为类型，你可以用大写字母开头，像下面这样：

```
Io> Ferrari := Car clone
==> Ferrari_0x9d085c8:
type = "Ferrari"

Io> Ferrari type
==> Ferrari

Io> Ferrari slotNames
==> list("type")
Io> ferrari slotNames
==> list()
Io>
```

注意，这里ferrari没有type槽，而Ferrari却有。我们采用简明的编程惯例而不是完整的语言特性来区分类型和实例。在其他方面，它们的行为是相同的。

在Ruby和Java中，类是用来创建对象的模板。bruce = Person.new这条语句从Person类创建一个新的Person对象。bruce和Person是完全不同的两个实体，一个是对象，一个是类。在Io中却不是这样。bruce := Person clone这条语句从Person原型创建一个名为bruce的复制品。bruce和Person都是对象。Person还是类型，因为它有type槽。而在其他方面，Person和bruce完全相同。下面，我们来看看具体行为。

3.2.3 方法

在Io中，你可以轻松地创建方法，就像下面这样：

```
Io> method("So, you've come for an argument." println)
==> method(
    "So, you've come for an argument." println
)
```

方法也是对象，就像所有其他类型的对象一样。你可以获取它的类型：

```
Io> method() type
==> Block
```

既然方法是对象，我们就可以把它赋值给一个槽：

```
Io> Car drive := method("Vroom" println)
==> method(
    "Vroom" println
)
```

如果某个槽是方法，调用这个槽会调用该方法：

```
Io> ferrari drive
Vroom
==> Vroom
```

信不信由你，你现在已经明白了Io的核心组织原则。回想一下，你知道Io的基本语法。你会定义类型和对象。通过把内容赋值给对象的槽的方式，你会把数据和行为添加到对象上。至于其他内容，就需要去研究库了。

我们再来多学习一点内容。你可以获取槽中的内容，无论它是变量还是方法，就像下面这样：

```
Io> ferrari getSlot("drive")
==> method(
    "Vroom" println
)
```

如果该槽不存在，getSlot会提供父对象的槽：

```
Io> ferrari getSlot("type")
==> Car
```

你可以获取对象的原型：

```
Io> ferrari proto
==>  Car_0x100473938:
  drive            = method(...)
  type             = "Car"

Io> Car proto
==>  Vehicle_0x1003b61f8:
  description      = "Something to take you far away"
  type             = "Vehicle"
```

这两个就是分别用来复制ferrari和Car的原型，而且出于方便考虑，这里还显示了它们的自定义槽。

Lobby是主命名空间，包含了所有的已命名对象。我们刚才在命令行中执行的所有赋值，以及另外几个对象，全部都在Lobby当中，如下所示：

```
Io> Lobby
==>  Object_0x1002184e0:
  Car              = Car_0x100473938
  Lobby            = Object_0x1002184e0
  Protos           = Object_0x1002184e0
  Vehicle          = Vehicle_0x1003b61f8
  exit             = method(...)
  ferrari          = Car_0x1004f43d0
  forward          = method(...)
```

你可以看到exit的实现、forward、Protos还有我们定义的其他东西。

原型编程范型看来十分清晰，下面就是这种范型的几条基本原则：

- 所有事物都是对象；
- 所有与对象的交互都是消息；
- 你要做的不是实例化类，而是复制那些叫做原型的对象；
- 对象会记住它的原型；
- 对象有槽；
- 槽包含对象（包括方法对象）；
- 消息返回槽中的值，或调用槽中的方法；
- 如果对象无法响应某消息，则它会把该消息发送给自己的原型。

差不多就是这些。你既可以看见、也可以改变任何槽或对象，所以能够进行一些相当复杂的元编程。不过，你首先必须理解更高级的构建元素——集合。

3.2.4　列表和映射

Io包含了几种类型的集合。列表（list）是任意类型对象的有序集合，所有列表的原型都是List对象。而Map对象是键值对的原型，称为映射（map），如同Ruby的散列表一样。我们可以像下面这样创建列表：

```
Io> toDos := list("find my car", "find Continuum Transfunctioner")
==> list("find my car", "find Continuum Transfunctioner")

Io> toDos size
==> 2

Io> toDos append("Find a present")
==> list("find my car", "find Continuum Transfunctioner", "Find a present")
```

Io在表示列表时有一种快捷方式：由Object对象提供的list方法。它能把传入方法的参数包装起来以形成列表。使用list方法，我们可以像下面这样方便地创建列表：

```
Io> list(1, 2, 3, 4)
==> list(1, 2, 3, 4)
```


如果想对列表进行数学运算，或把列表用作其他数据类型（如栈）处理，List对象也提供了各种简便方法：

```
Io> list(1, 2, 3, 4) average
==> 2.5

Io> list(1, 2, 3, 4) sum
==> 10

Io> list(1, 2, 3) at(1)
==> 2

Io> list(1, 2, 3) append(4)
==> list(1, 2, 3, 4)

Io> list(1, 2, 3) pop
==> 3

Io> list(1, 2, 3) prepend(0)
==> list(0, 1, 2, 3)

Io> list() isEmpty
==> true
```

Io的另一种主要集合是Map对象。Io的映射就像Ruby的散列表。由于映射没有语法糖，你必须用API操作它，就像下面这样：

```
Io> elvis := Map clone
==>  Map_0x115f580:

Io> elvis atPut("home", "Graceland")
==>  Map_0x115f580:

Io> elvis at("home")
==> Graceland
```

```
Io> elvis atPut("style", "rock and roll")
==>  Map_0x115f580:

Io> elvis asObject
==>  Object_0x11c1d90:
  home             = "Graceland"
  style            = "rock and roll"

Io> elvis asList
==> list(list("style", "rock and roll"), list("home", "Graceland"))

Io> elvis keys
==> list("style", "home")

Io> elvis size
==> 2
```

仔细琢磨一下，你会发现散列表的结构和Io对象很像——散列表的键就是一个个绑定了值的槽。因此，像Io映射那样把这些槽的组合方便快捷地转化为对象就非常有用[①]。

既然知道了这些基本集合，你当然想用它们做点什么。这少不了要介绍Io的控制结构。但作为控制结构的基础，必须先介绍一下布尔值。

3.2.5　true、false、nil以及单例

Io的条件语句和其他面向对象语言非常相似。下面给出了一些条件语句：

```
Io> 4 < 5
==> true
Io> 4 <= 3
==> false
Io> true and false
==> false
Io> true and true
==> true
Io> true or true
==> true
Io> true or false
==> true
Io> 4 < 5 and 6 > 7
==> false
Io> true and 6
==> true
Io> true and 0
==> true
```

这些都没什么难的。不过要注意：和Ruby一样0是true，而不像C语言那样是false。那么，true又是什么？

① 即上面代码的asObject方法。

```
Io> true proto
==>  Object_0x200490:
                    = Object_()
  !=                = Object_!=()
...

Io> true clone
==> true
Io> false clone
==> false
Io> nil clone
==> nil
```

这可就有意思了！true、false和nil都是单例（singleton）[①]，对它们进行复制，返回的只是单例对象的值。想做到这一点并不难，你可以像下面这样创建自己的单例：

```
Io> Highlander := Object clone
==>  Highlander_0x378920:
  type              = "Highlander"

Io> Highlander clone := Highlander
==>  Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"
```

我们只不过重定义了clone方法，让它返回Highlander对象自身，而不是像往常那样，让请求沿着对象树向上传递，最终到达Object对象。现在，当你使用Highlander对象时，它的行为如下所示：

```
Io> Highlander clone
==>  Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"
Io> fred := Highlander clone
==>  Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"

Io> mike := Highlander clone
==>  Highlander_0x378920:
  clone             = Highlander_0x378920
  type              = "Highlander"

Io> fred == mike
==> true
```

① 单例模式是一种很常见的软件设计模式，它保证某个类型只有一个对象实例存在。这样做的目的是，很多时候系统仅仅需要一个全局对象，而不是每次都返回一个新的对象实例。该模式可用在界面、数据库、服务器配置等诸多方面。

这里我们看到，Highlander对象的两个复制品是相等的。但如果不用单例，这种情况一般不会发生：

```
Io> one := Object clone
==>  Object_0x356d00:

Io> two := Object clone
==>  Object_0x31eb60:

Io> one == two
==> false
```

这样，就可以做到仅有一个Highlander对象。虽然，Io有时会给你设下一些陷阱，但对于单例而言，Io的解决方法虽有些出人意表，却仍不失为一种简明优雅的方法。在这一小节当中，我们经历了大量信息的狂轰滥炸，然而，我们也充分明白了如何去完成一些基本任务，比如改变对象的clone方法以实现单例。

尽管如此，你仍要多加小心。不管对Io是爱是恨，你都无法否认，它是一门有趣的语言。和Ruby一样，Io也能让人爱恨交加。你几乎可以改变任意对象上的任意一个槽，甚至那些定义这门语言的对象也概莫能外。比如下面这行代码，你可能就不会有随便尝试的意愿：

```
Object clone := "hosed"
```

因为你覆盖了Object上的clone方法，所以你从此无法再创建对象了。而且，这种情况无法修复，你只有终止进程这条路可走，但同时，你也可以用它获得一些令人神驰目眩的行为。既然你完全掌握运算符和组成对象的各个槽，实现领域特定语言（domain-specific language）只需要寥寥数行简短且漂亮的代码就可以了。在你重温今天所学的知识之前，我们先来聆听Io发明者的真知灼见。

3.2.6　Steve Dekorte访谈录

Steve Dekorte是旧金山区的一名独立咨询师。他于2002年发明了Io。我有幸就他创建Io的经验体会进行了采访。

Bruce: 你为什么要开发Io?

Steve: 2002年，我朋友Dru Nelson发明了一门叫做Cel（受Self语言[①]启发）的语言，要我们对它的实现提一些反馈意见。我觉得我对这门语言的运行机制不怎么熟悉，提不出什么有价值的意见，于是我着手写了一门小巧的语言，以便更好地理解Cel的各种机制。这门小巧的语言发展成了Io。

Bruce: 你最钟爱它哪一点呢?

Steve: 我钟爱它简洁一致的语法和语义，这有助于理解代码做了些什么。你可以迅速学会

① Self语言也是一门基于原型的语言。它由David Ungar和Randall Smith于1986年在施乐帕洛阿尔托研究中心（Xerox PARC）工作时设计出来，并在一年后两人去斯坦福大学工作时实现了首个编译器。Self语言影响了后来的不少语言，包括JavaScript、Io、Cel、Lisaac等。

这门语言的基础。我本人的记性不太好，经常忘记C语言的语法和那些古怪的语义规则，因此不得不去查阅它们。（编者按：Steve用C语言实现了Io。）而我在使用Io的时候，就不必老惦记着查阅这种事了。

比如说，你只需看着代码（如`people select(age > 20) map(address) println`），就能较为清楚地了解代码做了些什么。它先是根据年龄筛选出人的列表，再获得他们的地址，最后把地址打印出来。

如果充分简化语义，它就会更具灵活性。你可以就此创作一些实现这门语言时尚未定义的东西。我想举个例子说明这点，电子游戏的解谜游戏预设了解题方法，但也有很多游戏没预设什么，而是开放式结局，那些结局开放的游戏更好玩些，因为你可以去做游戏设计者从未想到过的事情。Io也同样如此。

3

有时，其他语言会开辟一些语法捷径，这产生了额外的转换规则。当你用某门语言编程时，你的脑子里必须有相应的语法转换器，语言越复杂，脑子里的转换器就会越多，转换器的工作越重，你的工作也就越重。

Bruce: Io有哪些局限？

Steve: Io的灵活性可能会让它面对很多常见用途时速度较慢。不过，Io在某些方面也有速度优势[如：协程（coroutine）、异步套接字、SIMD①支持等]，这也使得Io相比于那些采用传统线程（每个套接字并发或非SIMD向量运算时产生）编写的程序要快得多，即便这程序是用C语言编写的，也同样如此。

Io较少的语法会使快速检查代码变得更加困难，对这一点，我也颇有微词。然而，我用Lisp也发现了同样问题，因此这也是可以理解的。额外的语法的确有利于速读代码。不过，新用户有时一开始会抱怨Io的语法实在太少，但在大多数情况下，他们都会渐渐喜欢上这一点。

Bruce: 在实际产品中，你见过的最特别的Io应用是什么？

Steve: 这些年来，我听说过各种各样关于Io的传闻，比如用在卫星上，用作路由配置语言，还用作电子游戏的脚本语言。皮克斯动画公司也用Io，他们还写了一条关于Io的博客文章。

第一天的学习真是忙碌，现在该休息一会儿了。你可以暂时停下脚步，把已学到的内容实践一下。

3.2.7 第一天我们学到了什么

你已体验过不少Io的有趣内容。你知道了Io的基本特征。Io这门原型语言有着非常简单的语法，你可以用这些语法去构建属于这门语言本身的全新基本元素。对Io来说，甚至连核心元素也不带有最简单的语法糖。在某些情况下，这种最简语法的方式会对你阅读代码的语法增加些许难度。

最简语法也有一些好处。既然能用语法做的事不多，你也就不必学习语法的各种特殊规则和

① 即单指令流多数据流（Single Instruction Multiple Data），是在同一时间对多个数据进行同一指令操作，从而实现并行化的技术。

例外规则。只要知道怎么读句子，就能够读懂这些语法。学习Io时，你的词汇量也会有所增加。

作为一名新学生，你的学习负担是大大减轻了：

- 理解几条语法规则；
- 理解消息；
- 理解原型；
- 理解库。

3.2.8　第一天自习

当你查找Io的背景资料时，搜索习题答案会稍微有些困难，因为Io的不同含义太多了。我建议你用Google搜索“Io languange”关键字。

找

- 一些Io的示例问题。
- 一个可解答问题的Io社区。
- 带有Io惯用法的风格指南。

答

- 对`1+1`求值，然后对`1 + "one"`求值。Io是强类型还是弱类型？用代码证实你的答案。
- 0是`true`还是`false`？空字符串是`true`还是`false`？`nil`是`true`还是`false`？用代码证实你的答案。
- 如何知道某个原型都支持哪些槽？
- `=`（等号）、`:=`（冒号等号）、`::=`（冒号冒号等号）之间有何区别？你会在什么情况下使用它们？

做

- 从文件中运行Io程序。
- 给定槽的名称，执行该槽中的代码。

花上点时间熟悉槽和原型。理解原型的运行方式。

3.3　第二天：香肠大王

回想一下Ferris Bueller吧。在那部电影里，Ferris Bueller这名出身于中产阶级的高中生，居然号称自己是芝加哥的香肠大王。这一幕也成了经典的装腔作势的桥段。正因为他懂得变通、不拘泥于陈规，所以免费在豪华饭店里美餐了一顿。如果你有Java背景并且喜欢Java，那么你会觉得不该如此——太过放纵未必是件好事。从Java社区看来，Bueller的那套把戏，还是丢到一边为好。不过到了Io当中，你有必要学会一些变通之道，这样才能充分利用Io的强大威力。如果你有开发Perl脚本背景，那么你大概会喜欢Bueller的把戏，因为这帮他达到了目的。如果你编程一向较为迅速，但代码略显杂乱无章，那么在用Io时，你就必须稍微收敛一下自己的散漫，融入一些章法。在第二天中，你将要了解到如何用Io的槽和消息构造核心行为。

3.3.1 条件和循环

Io的所有条件语句在实现时都没用到什么语法糖。你会发现，它们易于理解和记忆，但读起来有点难度，因为其中没有多少语法信息。写个无限循环很简单，你可以按Ctrl+C中断它：

```
Io> loop("getting dizzy..." println)
getting dizzy...
getting dizzy...
...
getting dizzy.^C
IoVM:
        Received signal. Setting interrupt flag.
...
```

在使用各种并发结构时，循环常常有其用武之地。不过一般来说，有条件限制的循环结构更为常用，比如while循环。while循环带有一个条件和一个用来求值的消息。记住，分号会把两个不同的消息连接起来：

```
Io> i := 1
==> 1
Io> while(i <= 11, i println; i = i + 1); "This one goes up to 11" println
1
2
...
10
11
This one goes up to 11
```

for循环也能做到同样的事。for循环带有一个计数器、一个初始值、一个终止值、一个可选的增量、以及一个带发送者[①]的消息。

```
Io> for(i, 1, 11, i println); "This one goes up to 11" println
1
2
...
10
11
This one goes up to 11
==> This one goes up to 11
```

也可以增加可选增量，就像下面这样：

```
Io> for(i, 1, 11, 2, i println); "This one goes up to 11" println
1
3
5
7
9
11
This one goes up to 11
==> This one goes up to 11
```

① 在稍后3.3.3节中将讲到。

实际上，你经常会用到任意数量的参数。你已经知道可选参数是for循环的第4个参数。然而，Io还允许你附加额外参数。这看似挺方便，但因为没有编译器帮你检查代码，所以必须多加留意：

```
Io> for(i, 1, 2, 1, i println, "extra argument")
1
2
==> 2
Io> for(i, 1, 2, i println, "extra argument")
2
==> extra argument
```

在第一种情况中，"extra argument"是名副其实的额外参数，根本用不上。而在第二种情况中，你去掉了可选增量参数，这实际上把后面的所有参数都向左移动了一个位置。"extra argument"现在变成了消息参数，而增量步长变成了返回i的i println。如果这行语句深埋在一个复杂的包当中，那Io的这项机制绝对会把你恶心死。有时候，拔个萝卜难免会带出泥。Io给了你自由，但有时自由也会害了你。

if控制结构是以函数的形式实现的，其形式如if(condition, true code, false code)。当条件为真时，该函数执行true code，否则执行false code：

```
Io> if(true, "It is true.", "It is false.")
==> It is true.
Io> if(false) then("It is true") else("It is false")
==> nil
Io> if(false) then("It is true." println) else("It is false." println)
It is false.
==> nil
```

在控制结构上花的工夫就这么多。下面，我们要用它们来开发自己的运算符。

3.3.2　运算符

像面向对象语言一样，很多原型语言也在运算符上用到了语法糖。它们是采用特殊形式的特殊方法，就像“+”和“/”这样。在Io中，你可以直接看到运算符表，如下所示：

```
Io> OperatorTable
==> OperatorTable_0x100296098:
Operators
  0   ? @ @@
  1   **
  2   % * /
  3   + -
  4   << >>
  5   < <= > >=
  6   != ==
  7   &
  8   ^
  9   |
  10  && and
  11  or ||
```

```
  12  ..
  13  %= &= *= += -= /= <<= >>= ^= |=
  14  return

Assign Operators
  ::= newSlot
  :=  setSlot
  =   updateSlot

To add a new operator: OperatorTable addOperator("+", 4)
    and implement the + message.
To add a new assign operator: OperatorTable addAssignOperator(
    "=", "updateSlot") and implement the updateSlot message.
```


你可以看到，赋值（assign）是另一种类型的运算符。运算符左边的数字代表该运算符的优先级，参数优先绑定到优先级靠近0的运算符上。从表中可看到“+”先于“==”求值，这点你应该能够猜到。但用“()”可以改变这种优先级。我们现在来定义一个异或运算符`xor`。如果只有一个参数为真，则`xor`返回`true`，否则返回`false`。首先，我们把这个运算符添加到运算符表中：

```
Io> OperatorTable addOperator("xor", 11)
==> OperatorTable_0x100296098:
Operators
  ...
  10  && and
  11  or xor ||
  12  ..
  ...
```

你可以在添加的那个优先级位置上看到新运算符。接下来，我们要对`true`和`false`两种情况实现`xor`方法：

```
Io> true xor := method(bool, if(bool, false, true))
==> method(bool,
    if(bool, false, true)
)
Io> false xor := method(bool, if(bool, true, false))
==> method(bool,
    if(bool, true, false)
)
```

这里为了保持概念简单而使用了暴力穷举法。我们实现的运算符行为准确无误，正如前面预期的那样：

```
Io> true xor true
==> false
Io> true xor false
==> true
Io> false xor true
==> true
Io> false xor false
==> false
```

综上所述，true xor true会被解析为true xor(true)。而之前运算符表那个方法，确定了新运算符的优先级顺序，以及优先级作用下的简化语法①。

赋值运算符自成一表，运行起来和其他运算符不太一样。它们会像消息那样运行。你可以在3.4节中见到一个实例。现在，我要讲到的运算符内容已全部讲完。下面，我们开始学习消息。通过消息，你能懂得如何实现自己的控制结构。

3.3.3　消息

在我写作本章的过程中，Io代码的一位贡献者帮助我克服了不少困难。他说："Bruce，Io中有个东西你非理解不可，因为几乎一切都是消息。"如果你看看Io代码，会发现除了注释符和参数之间的逗号外，一切事物都是消息。没错，一切事物！学好Io意味着不仅仅学会通过调用来操作这些消息。在这门语言中，消息反射是一项至关重要的能力。你可以查询任何消息的任何特性，再对它们执行适当的操作。

一个消息由三部分组成：发送者（sender）、目标（target）和参数(arguments)。在Io中，消息由发送者发送至目标，然后由目标执行该消息。

你可以用call方法访问任何消息的元信息（meta information）。下面，我们创建两个对象：一个是获得消息的邮局对象postOffice，一个是发送消息的寄信人对象mailer。

```
Io> postOffice := Object clone
==>  Object_0x100444b38:

Io> postOffice packageSender := method(call sender)
==> method(
    call sender
)
```

下面，创建发送消息的寄信人：

```
Io> mailer := Object clone
==>  Object_0x1005bfda0:

Io> mailer deliver := method(postOffice packageSender)
==> method(
    postOffice packageSender
)
```

它有一个槽——deliver。该槽发送一个packageSender消息给postOffice。现在，我们有了一个可以发送消息的mailer对象：

```
Io> mailer deliver
==>  Object_0x1005bfda0:
  deliver          = method(...)
```

① 所谓优先级作用下的简化语法，指的是类似1+2*3这样的形式，而对应的未简化语法（即带括号的形式）为1+(2*3)。虽然简化后代码较短，但为了清晰起见，一般的编程惯例提倡通过括号显式地表明运算顺序。

也就是说，deliver方法是发送消息的对象。我们还可以获取目标，像下面这样：

```
Io> postOffice messageTarget := method(call target)
==> method(
    call target
)
Io> postOffice messageTarget
==>  Object_0x1004ce658:
  messageTarget    = method(...)
  packageSender    = method(...)
```

这非常简单。你从槽名可以看出，这里的目标是邮局postOffice。我们还能获取原消息名和参数，如下所示：


```
Io> postOffice messageArgs := method(call message arguments)
==> method(
    call message arguments
)
Io> postOffice messageName := method(call message name)
==> method(
    call message name
)
Io> postOffice messageArgs("one", 2, :three)
==> list("one", 2, : three)
Io> postOffice messageName
==> messageName
```

可见，Io有不少能用来进行消息反射的方法。接下来的问题是：Io何时计算消息？

大部分语言都将参数作为栈上的值传递。举例来说，Java首先计算参数的每个值，然后把这些值放到栈上。Io就不这样。Io传递的是消息本身和上下文，再由接受者对消息求值。实际上，你可以用消息实现控制结构。回想一下Io的if，其形式是if(booleanExpression, trueBlock, falseBlock)。假如你现在想再实现一个unless，实现方法可能会像下面这样：

io/unless.io

```
unless := method(
    (call sender doMessage(call message argAt(0))) ifFalse(
    call sender doMessage(call message argAt(1))) ifTrue(
    call sender doMessage(call message argAt(2)))
)

unless(1 == 2, write("One is not two\n"), write("one is two\n"))
```

这个小例子很漂亮，所以请认真阅读它。你可以将doMessage想象成类似于Ruby的eval，但更基础一些。Ruby的eval把字符串求值为代码，而doMessage可执行任意消息。Io会对消息参数进行解释，但会延迟绑定和执行。在典型的面向对象语言中，解释器或编译器可能会计算所有参数，包括两者的代码块，然后把它们的返回值放到栈上。但在Io中，事情完全不是这样。

假如对象westley发送消息princessButtercup unless(trueLove, ("It is false" println),("It is true" println))，则其流程如下所示：

(1) 对象westley发送上一条消息；

(2) Io取得经过解释的消息以及上下文（该调用的发送者、目标、消息），并将其放到栈上；

(3) 现在，princessButtercup对消息求值。它没有unless槽，因此Io沿着原型链（prototype chain）向上搜索，直至找到unless消息为止；

(4) Io开始执行unless消息。首先，Io执行call sender doMessage(call message argAt(0))。这句代码可化简为westley trueLove。如果你看过《公主新娘》[1]这部电影，你会知道，westley有一个叫trueLove的槽，并且该槽的值为true；

(5) 既然这消息的值不是false，那我们执行第三个代码块，该代码块化简为westley ("It is true" println)。

我们利用了一个事实：Io不是执行参数来计算实现unless控制结构的返回值的。这一思想极其强大。迄今为止，你已看到了反射等式的一边：带消息反射的行为。该等式的另一边是状态。下面，我们会和对象的槽一起观察状态。

3.3.4　反射

Io提供了一组简单的方法，通过这些方法，你很容易看清对象的槽都做了些什么。下面的代码创建了两个对象，并沿着原型链层层向上推进，最终找到一个名为ancestors的方法：

io/animals.io

```
Object ancestors := method(
        prototype := self proto
        if(prototype != Object,
        writeln("Slots of ", prototype type, "\n---------------")
        prototype slotNames foreach(slotName, writeln(slotName))
        writeln
        prototype ancestors))


Animal := Object clone
Animal speak := method(
            "ambiguous animal noise" println)
Duck := Animal clone
Duck speak := method(
            "quack" println)

Duck walk := method(
            "waddle" println)

disco := Duck clone
disco ancestors
```

① *The Princess Bride*，讲述了一对青年男女追求真爱的冒险过程，其中男主人公的名字为Westley。——编者注

这段代码不算太复杂。首先，我们创建了一个Animal原型，并用它创建了一个带speak方法的Duck实例。Duck也是disco的原型。ancestors方法打印该对象原型的槽，然后又在原型上调用ancestors方法。记住，一个对象可以有好几个原型，但这里我们只处理一个原型的情况。为了节省篇幅，我们在它打印出Object原型的所有槽之前中止了递归。运行io animals.io，输出如下：

```
Slots of Duck
---------------
speak
walk
type

Slots of Animal
---------------
speak
type
```

这些输出都在我们意料之内。每个对象都有原型，而这些原型也都是带槽的对象。在Io中，处理反射分为两部分。在邮局的那个例子中，你见到了消息反射。对象反射的意思是，处理对象和对象的槽。这些都和类扯不上半点关系。

3.3.5　第二天我们学到了什么

如果你跟得上进度，第二天该算是突飞猛进的一天。你对Io的了解应达到在文档的些许帮助下能完成基本任务的水平。你学会了如何进行判断、定义方法、使用数据结构、使用基本控制结构。在下面的练习中，Io会接受我们的考验。一定要把Io彻底弄明白。否则我们后面进一步讨论Io，包括元编程和并发空间等问题时，你绝对会后悔当初怎么没完全掌握这些基本知识。

3.3.6　第二天自习

做

- 斐波那契数列以两个1开始，每一个数都是之前两个数之和：1，1，2，3，5，8，13，21，以此类推。写一个计算第*n*个斐波那契数的程序。这里fib(1)会得到1，fib(4)会得到3。如果你能用递归和循环两种方法写出来，我将给你额外加分。
- 在分母为0的情况下，如何让运算符/返回0？
- 写一个程序，把二维数组中的所有数相加。
- 对列表增加一个名为myAverage的槽，以计算列表中所有数字的平均值。如果列表中没有数字会发生什么？（加分题：如果列表中有任何一项不是数字，则产生一个Io异常。）
- 对二维列表写一个原型。该原型的dim(x, y)方法可为一个包含y个列表的列表分配内存，其中每个列表都有x个元素，set(x, y)方法可设置列表中的值，get(x, y)方法可返回列表中的值。

- 加分题：写一个转置方法，使得原列表上的`matrix get(x, y)`与转置后列表的`(new_matrix get(y, x))`相等。
- 把矩阵写入文件，并从文件中读取矩阵。
- 写一个程序，提供10次尝试机会，猜一个1 ~ 100之间的随机数。如果你愿意的话，可以在第一次猜测之后，提示猜大了还是猜小了。

3.4　第三天：花车游行和各种奇妙经历

初次接触Io的那段日子让我灰心丧气、不知所措，但过了两三周之后，我居然感觉自己像未经世事的少女一般，一想到Io将带我体验怎样的奇妙之旅，就会情不自禁地低声轻笑起来。这就像Ferris在新闻上、在棒球场中、在花车游行时频频曝光——总会在你意想不到的地方出现。说到底，我真真切切地通过Io，体验到了我想要的感觉——用一门语言改变自己思维方式的感觉。

3.4.1　领域特定语言

几乎每一位深入研究过Io的人，都会对它在DSL方面的强大能力赞不绝口。Jeremy Tregunna是Io代码的核心贡献者，他告诉我，用Io实现C语言的一个子集仅需约40行代码！不过对我们来说，这个例子有点过于深奥了，因此Jeremy拿出了另一个压箱底的宝贝送给我们——实现了一种有趣的电话号码语法的API。

比如，你想用以下形式表示电话号码：

```
{
    "Bob Smith": "5195551212",
    "Mary Walsh": "4162223434"
}
```

管理这样一个列表的方法有很多，其中最容易想到的有两种：对列表进行语法分析、对列表进行解释。语法分析意味着写一个识别列表语法中不同元素的程序，然后就能把列表代码组织成Io可理解的结构。但我们今天不会讨论这个问题。与之相比，把列表代码解释为Io散列表的形式要有趣得多。为了能做到这点，我们必须对Io做些改动。改动完毕之后，Io就会认为上面的列表的语法是正确的，并且根据该列表构建出散列表来！

下面就是Jeremy在Chris Kappler（他用当前版本的Io，更新了原先的旧代码）的帮助下，解决这一问题的办法：

io/phonebook.io

```
OperatorTable addAssignOperator(":", "atPutNumber")
curlyBrackets := method(
  r := Map clone
  call message arguments foreach(arg,
       r doMessage(arg)
       )
  r
```

```
)
Map atPutNumber := method(
  self atPut(
       call evalArgAt(0) asMutable removePrefix("\"") removeSuffix("\""),
       call evalArgAt(1))
)
s := File with("phonebook.txt") openForReading contents
phoneNumbers := doString(s)
phoneNumbers keys   println
phoneNumbers values println
```

这些代码比我们先前见过的代码都要稍微复杂些，但你已经知道了其中的基本单元都是什么。下面让我们来解构它：


```
OperatorTable addAssignOperator(":", "atPutNumber")
```

代码的第一行，把一个运算符添加到了Io的赋值运算符表中。现在，只要在Io代码中遇到“:”，Io都会把它转换为atPutNumber。这里你要知道，第一个参数是名称（因此它是字符串），第二个参数是值。这样一来，key : value就将转换为atPutNumber("key", value)。继续看下面的代码：

```
curlyBrackets := method(
  r := Map clone
  call message arguments foreach(arg,
       r doMessage(arg)
       )
  r
)
```

只要Io代码遇到了大括号（{}），转换程序就会调用这个curlyBrackets方法。在该方法中，我们创建了一个空映射。然后，我们对每个参数执行call message arguments foreach (arg, r doMessage(arg))。这行代码真的是太浓缩了！我们把它分解开来看看。

从左到右看，先是执行call message，它正是大括号中的那部分代码。然后，用foreach迭代遍历列表中的每一条电话号码。对于每一条中的名字和号码，执行r doMessage(arg)。例如，第一条电话号码将执行为r "Bob Smith": "5195551212"。由于:在操作符表中是atPutNumber，因此将执行r atPutNumber("Bob Smith", "5195551212")。下面我们来看看下面的代码：

```
Map atPutNumber := method(
  self atPut(
       call evalArgAt(0) asMutable removePrefix("\"") removeSuffix("\""),
       call evalArgAt(1))
)
```

记住，key : value会转换为atPutNumber("key", value)。在我们的例子中，键总是字符串，所以要去掉开头和结尾的两个引号。你可以看到，atPutNumber只是在目标（即self）上调用了atPut，把第一个参数的引号去掉。由于消息是不可变的（immutable），因此为了去掉引号，必须把它转化为一个可变值。

你可以使用如下所示代码：

```
s := File with("phonebook.txt") openForReading contents
phoneNumbers := doString(s)
phoneNumbers keys   println
phoneNumbers values println
```

理解Io语法不是什么大不了的事。你必须要去理解的是，在库里到底发生了些什么。在上面的例子中，你见到了几个新的库。doString把我们的电话号码簿求值为Io代码，File是与文件交互的原型，with指定了文件名并返回一个文件对象，openForReading打开该文件并且也返回该文件对象，而contents返回该文件的内容。把它们组合在一起，这些代码就可以读取电话簿并将其求值为Io代码。

然后，大括号定义了一个映射，映射中的每一行"string1" : "string2"都会转化为map atPut("string1", "string2")，电话号码的散列表就这样产生了。这样一来，在Io当中，由于你可以随心所欲地把运算符重定义为组成DSL的符号，因此，构建称心如意的DSL也就手到擒来了。

现在，你逐渐开始明白如何改变Io的语法。那么，你又该如何去动态改变Io的行为呢？在下一小节中，我们会讲解这个主题。

3.4.2　Io的method_missing

先复习一下控制流。Object原型包含了处理消息传递的一切行为。当你把消息发送给对象的时候，对象将完成下列事情：

(1) 计算所有参数，这些参数其实就是消息；

(2) 获取消息的名称、目标和发送者；

(3) 尝试用目标上的消息名称读取槽；

(4) 如果槽存在，返回其数据或触发其包含的方法；

(5) 如果槽不存在，则把消息转发给它的原型。

这些是Io的基本继承机制。一般来说，它们不会把你弄得晕头转向。

但在某些情况下，你还是会晕头转向。就像Ruby的method_missing那样，你也可以用Io的forward消息做到同样的事，但这样做的风险要更高一些。Io没有类，所以改变forward也将改变从Object获得基本行为的方式。这有点像走高空钢丝时同时抛接三柄小斧子。如果这样你都能安然无恙，那真是神乎其技了。我们这就来学习这样的技巧。

XML是对数据进行结构化的绝妙方式，但却有着令人作呕的语法。为了摆脱这语法，你可以写个程序，用Io代码来表示XML数据。

假如你想把下面的数据：

```
<body>
<p>
This is a simple paragraph.
</p>
</body>
```

表示成这样：

```
body(
    p("This is a simple paragraph.")
)
```

我们把这种新语言称作LispML。我们将用Io的`forward`处理这门语言，就像处理不存在的方法（missing method）一样。下面给出代码：

io/builder.io

```
Builder := Object clone

Builder forward := method(
  writeln("<", call message name, ">")
  call message arguments foreach(
        arg,
        content := self doMessage(arg);
        if(content type == "Sequence", writeln(content)))
  writeln("</", call message name, ">"))

Builder  ul(
        li("Io"),
        li("Lua"),
        li("JavaScript"))
```

剖析一下这段代码。`Builder`原型是代码的核心部分。它覆盖了`forward`，使其可接收任意方法。首先，它打印一个开始标签（"<"）。然后，我们用到了一点消息反射。如果这消息是个字符串，Io会把它识别为序列（sequence），所以`Builder`会把该字符串打印出来（打印时不带引号）。最后，`Builder`打印一个结束标签（">"）。

输出与你想象的结果完全相同：

```
<ul>
<li>
Io
</li>
<li>
Lua
</li>
<li>
JavaScript
</li>
</ul>
```

其实，LispML未必比传统的XML有多大程度的提高，但这例子还是很有指导意义。你可以完全改变一个Io原型的继承运行机制。`Builder`原型的所有实例都将拥有同样的行为。这样一来，通过定义自己的`Object`原型，并以这个新对象为基础创建其他原型，你就可以用Io语法创建一门和Io行为截然不同的新语言。甚至，你还可以覆盖`Object`以复制新对象。

3.4.3　并发

Io有非常出色的并发库，其主要组成部分包括协程、actor和future。

1. 协程

协程是并发的基础。它提供了进程的自动挂起和恢复执行的机制。你可以把协程想象为带有多个入口和出口的函数。每次`yield`都会自动挂起当前进程，并把控制权转到另一进程当中。通过在消息前加上`@`或`@@`，你可以异步触发消息，前者将返回future（稍后详述），后者会返回`nil`，并在其自身线程中触发消息。举个例子，考虑下面的程序：

io/coroutine.io

```
vizzini := Object clone
vizzini talk := method(
            "Fezzik, are there rocks ahead?" println
            yield
            "No more rhymes now, I mean it." println
             yield)

fezzik := Object clone

fezzik rhyme := method(
                        yield
            "If there are, we'll all be dead." println
            yield
            "Anybody want a peanut?" println)

vizzini @@talk; fezzik @@rhyme

Coroutine currentCoroutine pause
```

`fezzik`和`vizzini`都带有协程，它们是彼此无关的`Object`实例。我们异步触发`talk`和`rhyme`方法。它们并发执行，用`yield`消息在指定时间段自动把控制权交给另一方法。最后一行的`pause`用来等待所有异步消息执行完毕，然后退出程序。协程在面对需要多任务合作的解决方案时表现完美。通过这个示例程序，两个需要彼此协作的进程能够轻易完成读诗（read poetry）任务，像下面这样：

```
batate$ io code/io/coroutine.io
Fezzik, are there rocks ahead?
If there are, we'll all be dead.
No more rhymes now, I mean it.
Anybody want a peanut?
Scheduler: nothing left to resume so we are exiting
```

Java和基于C的语言使用的并发哲学叫做抢占式多任务（preemptive multitasking）。当你把这种并发策略和可变状态的对象结合使用时，程序会变得一团糟，因为我们难以预测它的行为，大多数团队现有的测试技术也几乎不可能对其进行调试。协程就不一样了。利用协程，应用程序可以在适当的时间放弃控制权。比如，分布式客户端可以在等待服务端响应时放弃控制权，工作者

进程也可以在处理完队列中的产品后暂停。

协程是组成更高级抽象概念（如actor）的基本元素。你可以把actor想成通用的并发原语，它可以发送消息、处理消息以及创建其他actor。actor接收到的消息是并发的。在Io中，actor把新到达的消息放到队列上，并用协程处理队列中的各个消息。

下面，我们就要来看一下actor。你不会相信用它写出来的代码居然能如此简单。

2. actor

和线程相比，actor有着巨大的理论优势。一个actor改变其自身的状态，并且通过严格控制的队列接触其他actor。而多个线程可以不受限制地改变彼此状态。线程容易受到被称做竞争条件的并发问题影响，在这种问题中，如果两个线程同时存取资源，可能导致不可预测的后果。

3

Io的动人之处就在于此，发送异步消息给任意对象就是actor，就这么简单。举一个小例子。首先，创建两个对象，分别叫做faster和slower：

```
Io> slower := Object clone
==>  Object_0x1004ebb18:

Io> faster := Object clone
==>  Object_0x100340b10:
```

然后，给这两个对象添加一个方法，叫做start：

```
Io> slower start := method(wait(2); writeln("slowly"))
==> method(
    wait(2); writeln("slowly")
)
Io> faster start := method(wait(1); writeln("quickly"))
==> method(
    wait(1); writeln("quickly")
)
```

我们可以通过简单的消息，在一行代码中顺序调用这两个方法，像下面这样：

```
Io> slower start; faster start
slowly
quickly
==> nil
```

它们会按顺序执行，因为一定是第一个消息结束之后，第二个消息才会开始。但我们可以通过在这两个消息之前加上@@，让对象在自己的线程中运行。这样做会立即返回nil：

```
Io> slower @@start; faster @@start; wait(3)
quickly
slowly
```

我们在最后加了一个wait，以便让所有线程在程序终止前执行完毕。这样做能获得非常棒的结果。我们同时运行了两个线程。只不过发送了异步消息给这两个对象，它们就全都变成了actor。

3. future

讲过future概念之后，关于并发的讨论就将告一段落。future是在异步调用消息时立即返回的一个结果对象。由于被调用的消息可能需要一段时间处理，因此到最终产生结果的时候，future

会变成这个结果值。如果在结果尚未产生时请求future的值，进程会阻塞直到产生结果为止。假如有这样一个执行时间很长的方法：

```
futureResult := URL with("http://google.com/") @fetch
```

我可以先执行这个方法，然后马上写一些做其他事情的代码，直到产生结果时再取回：

```
writeln("我们可以在后台运行网页抓取的同时做一些别的事情……")
// ...
```

这种情况下，就可以使用future值：

```
writeln("这里将阻塞，直到结果产生为止")
//上面这行代码会立刻执行

writeln("fetched ", futureResult size, " bytes")
//上面这行代码会阻塞，直到计算完毕

//然后Io打印出结果值
==> 1955
```

`futureResult`这段代码片断会立即返回一个future对象。在Io中，future并不是代理实现！future会阻塞到我们可获得结果对象时为止。future的值一开始是个`Future`对象，但等到结果产生之后，所有该future值的实例都会指向结果对象，命令行将会把最后一条返回语句的字符串值打印出来。

Io的future还会提供死锁自动检测机制。这可谓画龙点睛之笔，而且它既容易理解，也便于使用。

现在你已经对Io的并发机制有所了解，那么也就打下了正确评价这门语言的良好基础。下面，我们就总结一下第三天的学习成果，以便你能把它们运用到实践当中。

3.4.4　第三天我们学到了什么

在本节中，你学会了如何用Io完成一些重要的事。首先，我们就原有的语法规则做了一些“变通”，并用大括号构建了新的散列语法。我们把一个运算符加到运算符表当中，并把它和散列表中的操作联系起来。然后，我们构建了一个XML生成器，使用`method_missing`打印出XML元素。

接下来，我们写了一些使用协程管理并发的代码。协程与Ruby、C、Java等语言中的并发不同，因为协程只能改变它们自身的状态。有了协程，也就拥有了更易于预测和理解的并发模型，并且遭遇可能成为瓶颈的阻塞状态的情况也会减少。

我们发送一些使原型成为actor的异步消息。除了改变消息语法之外，不必做任何事情。最后，简单介绍了一下future和它在Io当中的运行方式。

3.4.5　第三天自习

做

- ❑ 改进本节生成的XML程序，增加空格以显示缩进结构。
- ❑ 创建一种使用括号的列表语法。

- 改进本节生成的XML程序，使其可处理属性：如果第一个参数是映射（用大括号语法），则为XML程序添加属性。例如：
  `book({"author": "Tate"}...)`将打印出`<book author="Tate">`：

3.5 趁热打铁

如果想学习如何使用基于原型的语言，Io能给予极大帮助。Io的语法极其简单，但语义却无比强大，就像Lisp一样。原型语言封装了数据和行为，就像面向对象编程语言一样。继承也比其他语言简单。Io没有类或模块。对象直接从其原型那里继承行为。

3.5.1 核心优势

原型语言通常具有良好的可塑性，你可以改变任意对象的任意槽。Io把这种灵活性发挥到了极致，你可以用它快速创建出你想要的语法。和Ruby一样，为了使Io具有如此强大的动态特性而做的某些权衡，相对应的也会让它在性能上有所损失，至少在单线程的情况下是这样。Io有强大的、现代的并发库，在很多场合成为一门优秀的并行处理语言。下面，我们就来看看Io最擅长哪些方面。

1. 占用空间

Io占用的空间很小。大多数Io应用程序的产品都是嵌入式系统。虽然Io这门语言个头小，但其功能强大且相当灵活，所以应用在嵌入式领域也就合情合理。Io的虚拟机也易于移植到不同的操作环境当中。

2. 简单

Io的语法极为简洁，学习它花不了多少时间。一旦你理解了它的核心语法，剩下的就是学习库是如何组织的了。就我的经验而言，在使用这门语言的头两个月中，我能够学会元编程，这速度已经相当快了。在Ruby中，我达到同样程度所花的时间要更长一点。而Java，我花了很多个月，才对元编程有所了解。

3. 灵活

Io的鸭子类型和自由度，能让你在任何时间改变任何对象的任何槽。Io这一自由宽松的特点，意味着你可以为了适应自己的应用环境而改变Io的基本规则。通过改变forward槽，随便在哪儿添加代理都非常容易。你也可以直接改变核心语言结构的槽，从而覆盖这些语言结构。你甚至可以快速创建出自己想要的语法。

4. 并发

与Java和Ruby不同的是，Io的并发结构非常与时俱进。actor、future和协程使得Io编写多线程应用程序要容易得多，而且写出来的程序更易于测试且拥有更出色的性能。Io也花了很多心思考虑可变数据和如何避免它们的问题。这些特征作为其核心库的组成部分，使我们轻松地认识到什么是健壮的并发模型。后面学习其他语言时，我们还会进一步巩固这些并发概念。你将在Scala、Erlang和Haskell中再见到actor。

3.5.2　不足之处

Io值得我们喜爱的地方很多，但不尽如人意之处同样不少，获得自由和灵活是要付出代价的。此外，由于在本书所有语言之中，Io社区是规模最小的，因此选它完成项目也要冒较大风险。下面看看Io带来的问题都有哪些。

1. 语法

Io几乎没什么语法糖。简单的语法就像一把双刃剑。一方面，简单的语法使Io清晰易懂，但为此付出的代价是，简单的语法常常使Io难以用简短的方式表达艰深的概念。换句话说，你会发现自己很容易明白某个程序如何用Io写出来，但与此同时，你也很难明白某个程序到底做了些什么。

我们可以拿Ruby作个对比。刚开始，你可能会觉得Ruby代码`array[-1]`让人迷惑不解，因为你并不理解这个语法糖：`-1`是数组最后一个元素的缩写形式。你还应该知道，`[]`是获取数组指定下标值的方法。一旦理解了这些概念，阅读代码就会轻松许多。但Io所做的权衡恰恰相反。刚开始不用知道太多，但在理解那些用语法糖表达起来轻松自如的概念时，或许就没那么容易了。

平衡语法糖是件困难的事，加得过多，会导致语言不易学习，难以记住使用方法；加得过少，在代码的表达方式上花的工夫就要多一些，而且很可能要费大量精力排错。说到底，语法不过是个喜好问题，Matz喜好丰富的语法糖，Steve就不这么想。

2. 社区

当前，Io社区的规模非常小。Io不像其他语言那样，总能找到合适的库。想找到Io程序员就更难了。通过使用设计良好的C接口（它可以和各种各样的语言交互），以及非常容易记忆的语法，这些问题可在某种程度上得到缓和。优秀的JavaScript程序员能在短时间内学会Io。但较小的社区的确是个缺点，也是阻碍功能强大的新语言不断发展的主要因素。今后，Io要么就作出一个杀手级的应用，吸引大家接受这门语言，或者是保持小众代言人的身份，继续陪太子读书。

3. 性能

脱离其他问题——如并发或应用程序设计——谈论性能通常是不明智的，但我必须指出，对于那些未经琢磨且为单线程服务的程序来说，Io有不少能拖慢其执行速度的特性。这个问题通过Io的并发结构可得到一定程度的缓解，但你仍然要记住Io在性能上的这种局限性。

3.5.3　最后思考

总之，我喜欢学习Io。它语法简单，占用空间也小，这都很吸引我。我还觉得，Io就像Lisp那样，同样具有以简单和灵活为主的哲学思想。Steve Dekorte在创造Io的过程中始终贯彻这一思想，造就了一门类似Lisp的原型语言。我认为，Io今后仍能在艰苦条件下不断发展。像Ferris Bueller一样，Io也有着充满光明但危机四伏的未来。

第 4 章 Prolog

Dibbs Sally。461-0192。

—— Raymond

Prolog这门语言有时特别聪明，有时又特别令人失望。只有当你知道如何提问时，你才会得到令人惊奇的答案。回想一下《雨人》[①]这部电影。我还记得片中的主角Raymond，他在前一晚读过一本电话簿后便可以背出Sally Dibbs的电话号码，而他当时翻电话簿的时候根本没有考虑是否需要记住这个号码。对于Raymond和Prolog，我经常问出这样两个分量等同的问题，“他是怎么知道的？”和“他怎么不知道？”。只要你能以正确的方式表达你的问题，那么他将是一个知识源泉。

Prolog与前两章谈到的编程语言有较大的不同。Io和Ruby被称为*命令式语言*（imperative language）。命令式语言就像是一本烹饪食谱，你需要精确地告诉计算机如何去完成一项工作。更高级别的命令式语言可能会给你带来更多杠杆效力，即将多个比较长的步骤合并为一个步骤。不过从根本上说，你其实是在列出原料的购物清单，并描述烤蛋糕的详细步骤。

在尝试编写本章之前，我花了几周时间学习和使用Prolog。在我不断加强理解的过程中，我读过多本教程。J.R.Fisher[②]的教程为我提供了一些难度很高的例子。而另外一本由A.Aaby[③]编写的入门教程则让我对结构和术语有了更清晰的理解，并且提供了大量实验。

Prolog是一门声明式编程语言（decalarative language）。你向Prolog提供一些事实（fact）和推论（inference），并让它为你推断。它更像是一名手艺高超的糕点师。说出你喜欢的蛋糕的特征，让糕点师挑选原料并按照你提供的规则为你烤出蛋糕。有了Prolog，你无需知道如何做，计算机会为你作出推断。

随意浏览一下互联网，你就能发现很多使用不到20行代码解决数独问题的例子，也能找到魔方以及诸多著名难题的解决方法，例如汉诺塔（Tower of Hanoi）（大约用了十几行代码）。Prolog

① *Rain Man*，DVD版，导演Barry Levinson。（1988年）。发行商：加利福尼亚州洛杉矶市米高梅电影制片公司（2000年）。（译者注：这部电影中译名为《雨人》。该片获得了61届奥斯卡最佳影片、最佳导演、最佳男主角和最佳原作剧本四项重要奖项。该片描述了一个手足情深的故事。一心想获得遗产的弟弟与患有自闭症的兄长在长途旅行中逐渐产生了对兄长的手足之情，由起初的疏远粗暴到最后的关心理解。）

② 参见http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html。——原书注

③ 参见http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html。——原书注

是最早成功的逻辑编程语言之一。你使用纯逻辑设置断言，Prolog判定它们是否为真。你可以在断言中留出空白，Prolog将尝试填充这些空白并使那些不完整的事实变为真。

4.1　关于 Prolog

Prolog是一门逻辑编程语言，它于1972年由Alain Colmerauer和Phillipe Roussel开发完成，在自然语言处理领域颇受欢迎。现在，从调度系统到专家系统，这门备受尊重的语言为各类问题提供了编程基础。你可以使用这门基于规则的语言来表达逻辑和提出问题。和SQL一样，Prolog基于数据库，但是其数据由逻辑规则和关系组成；和SQL一样，Prolog包含两个部分：一部分用于描述数据，而另一部分则用于查询数据。在Prolog中，数据以逻辑规则的形式存在，下面是基本构建单元。

- *事实*。事实是关于真实世界的基本断言。（Babe是一头猪，猪喜欢泥巴。）
- *规则*。规则是关于真实世界中一些事实的推论。（如果一个动物是猪，那么它喜欢泥巴。）
- *查询*。查询是关于真实世界的一个问题。（Babe喜欢泥巴吗？）

事实和规则被放入一个*知识库*（knowledge base）。Prolog编译器将这个知识库编译成一种适于高效查询的形式。当我们学习这些例子的时候，你可以使用Prolog表达知识库。然后你就可以直接检索数据，也可以使用Prolog将多个规则串联在一起来得到一些你可能不知道的事情。

关于Prolog的背景介绍已经足够多了。让我们开始正式的学习吧。

4.2　第一天：一名优秀的司机

在《雨人》这部影片中，Raymond告诉他的弟弟他是名优秀的司机，意思是他可以在停车场内以每小时5英里的速度驾驶一辆汽车。他可以使用汽车的所有主要部件，方向盘、刹车以及油门踏板，只是使用的环境很有限。这就是你今天的目标。我们将使用Prolog描述一些事实，编写一些规则，并且进行一些基本的查询。和Io一样，Prolog是一门语法极其简单的语言。你很快就可以学会其语法规则。当你用有趣的方法将其概念分层，真正的乐趣才开始。 如果这是你第一次使用Prolog，我可以肯定地告诉你，你要么改变思考方式，要么你的学习将以失败告终。我们将在第二天深入分析这一点。

首先最重要的是安装一个可用的Prolog实现版本。我在本书中使用的是GNU Prolog，版本号为1.3.1。注意，Prolog方言之间可能彼此不同。我会尽力停留在共同点上，不过如果你选择一个不同的Prolog版本，那就需要做一点功课了，了解一下你使用的方言有何不同。不管你选择了什么版本，这里要告诉你的是如何使用它。

4.2.1　基本概况

在一些语言中，大写字母如何使用完全由程序员自行决定。不过在Prolog中，第一个字母的大小写是有着重要意义的，如果一个词以小写字母开头，它就是一个*原子*（atom）——一个类似

Ruby符号（symbol）的固定值，如果一个词以大写字母或下划线开头，那么它就是一个变量。变量的值可以改变，原子则不能。让我们用一些事实来构建一个简单的知识库吧。把下面内容敲入到一个编辑器中：

prolog/friends.pl

```
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).

friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

上述文件是由事实和规则组成的知识库。前三行语句是事实，最后一行语句是一个规则。事实是我们对这个世界直接观察的结果。规则是关于现实世界的逻辑推论。现在，注意前三行语句。其中每一行都是一个事实。wallace，grommit和wendolene都是原子。你可以这样读出它们：wallace喜欢cheese，grommit喜欢cheese，wendolene喜欢sheep。让这些事实开始工作吧。

启动Prolog解释器。如果你用的是GNU Prolog，输入命令gprolog。然后输入下面内容以加载文件：

```
| ?- ['friends.pl'].
compiling /Users/batate/prag/Book/code/prolog/friends.pl for byte code...
/Users/batate/prag/Book/code/prolog/friends.pl compiled, 4 lines read -
997 bytes written, 11 ms

yes

| ?-
```

除非Prolog在等待一个中间结果，否则它都会用yes或no作出回应。在这个例子中，文件加载成功，所以解析器返回yes。我们可以开始问一些问题了。最基本的问题是一些关于事实的yes和no的问题。比如：

```
| ?- likes(wallace, sheep).

no
| ?- likes(grommit, cheese).

yes
```

这些问题都非常直观。wallace喜欢sheep吗？（No.）grommit喜欢cheese吗？（Yes.）好像没有什么吸引力：Prolog仅仅是鹦鹉学舌般地将事实重新呈现给你。当你开始加入一些逻辑时，它才会变得更为精彩。让我们看看一些推论吧。

4.2.2 基本推论和变量

下面来测试friend规则：

```
| ?- friend(wallace, wallace).

no
```

这样，Prolog就根据设置的规则来回答yes或no的问题。这里远比外表看起来的有深度。再看一下friend规则：

在规则中，如果X是Y的朋友，那么X就不可能与Y相同。看看:-右边的第一部分吧，这部分被称为一个子目标（subgoal）。\+执行逻辑取反操作，这样\+(X=Y)的意思就是X不等于Y。

再来做一些查询：

```
| ?- friend(grommit, wallace).

yes
| ?- friend(wallace, grommit).

yes
```

在英语中，如果可以证明X喜欢某个Z并且Y也喜欢同一个Z，那么X就是Y的朋友。wallace和grommit都喜欢cheese，所以这些查询都返回yes。

我们来深入分析一下代码。在这些查询中，X不等于Y，满足第一个子目标。查询将使用第二个和第三个子目标：likes(X, Z)和likes(Y, Z)。grommit和wallace都喜欢cheese，所以又满足了第二个和第三个子目标。尝试另外一个查询：

```
| ?- friend(wendolene, grommit).

no
```

在这个例子中，Prolog会尝试几组可能的X、Y和Z值：

- wendolene、grommit和cheese
- wendolene、grommit和sheep

两种组合都无法同时满足两个目标，即wendolene喜欢Z并且grommit也喜欢Z。因为不存在这样的组合，所以逻辑引擎报告no，即它们不是朋友。

我们来正式介绍一下这个术语。

```
friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

上述代码是一个具有三个变量X、Y和Z的Prolog规则。我们把这个规则称作friend/2，即有两个参数的friend规则的缩写。这个规则拥有三个用逗号分隔的子目标。当所有子目标都为真时，这个规则才为真。所以我们这个规则的含义是：如果X与Y不等同且X和Y都喜欢同一个Z，那么X是Y的朋友。

4.2.3　填空

我们用Prolog回答了一些yes或no的问题，不过我们能做的可不止这些。在这一节中，我们将使用逻辑引擎为一个查询找出所有可能的匹配。要做到这一点，你需要在查询中指定一个变量。

考虑下面的知识库：

prolog/food.pl

```
food_type(velveeta, cheese).
```

```
food_type(ritz, cracker).
food_type(spam, meat).
food_type(sausage, meat).
food_type(jolt, soda).
food_type(twinkie, dessert).

flavor(sweet, dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).
```

我们给出了一些事实。一些诸如food_type(velveeta, cheese)，意思是一种食物具有一定的类型。另外一些诸如flavor(sweet, dessert)，意思是一种食物类型具有特有的味道。最后，我们给出了一个名为food_flavor的规则，它可推断出食物的味道。如果食物X属于Z类食物且Z也具有特有味道Y，则食物X具有food_flavor Y。编译这段代码：


```
| ?- ['code/prolog/food.pl'].
compiling /Users/batate/prag/Book/code/prolog/food.pl for byte code...
/Users/batate/prag/Book/code/prolog/food.pl compiled,
12 lines read - 1557 bytes written, 15 ms

(1 ms) yes
```

问一些问题：

```
| ?- food_type(What, meat).

What = spam ? ;

What = sausage ? ;

no
```

现在很有趣。我们请求Prolog，“找出一些满足查询food_type(What, meat)的值。”Prolog找到了一个spam。输入;，请求Prolog找出下一个，它返回了sausage。由于这些查询依赖基本事实，所以值很容易找到。接下来，请求另一个值，Prolog回答no。这个行为可能稍有些不一致。为方便起见，如果在剩余部分中Prolog检测不到其他可选项，你将看到一个yes。如果Prolog在未经更多计算的情况下不能立刻断定是否还有更多选项，那么它将提示你查询下一个并返回no。这个特性真的很方便。如果Prolog可以尽早给你提供信息的话，它就会这么做。再试几个查询：

```
    | ?- food_flavor(sausage, sweet).

    no

| ?- flavor(sweet, What).

What = dessert ? ;
```

```
What = soda ? ;

yes
```

不，sausage不是sweet的。什么种类的食物味道是sweet？dessert和soda。这些全部是事实。不过你也可以让Prolog把各种事实串到一起：

```
| ?- food_flavor(What, savory).

What = velveeta ? ;

What = spam ? ;

What = sausage ? ;

no
```

记住，food_flavor(X, Y)是一个规则，不是一个事实。我们请求Prolog找出满足“什么食物具有savory味道？”这个查询的所有可能值。Prolog必须将关于食物、类型和味道的基本事实联系在一起才能得出最终结论。逻辑引擎需要遍历所有使目标为真的可能组合。

1. 地图着色

下面用同样的思路来进行地图着色。为了更深入地观察Prolog，我们采用了这个例子。这里要给美国东南部的地图着色，填充图4-1中所展示的各州。我们不想两个接壤的州具有相同颜色。

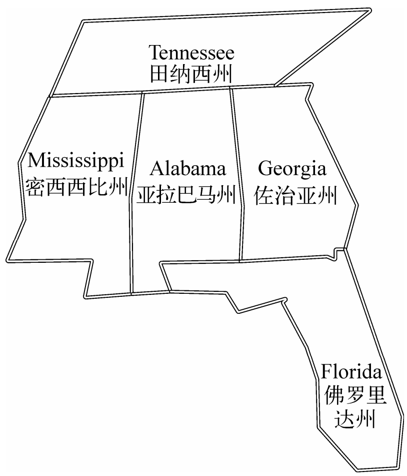


图4-1　部分东南部州的地图

对这些简单的事实进行编码：

prolog/map.pl

```
different(red, green). different(red, blue).
different(green, red). different(green, blue).
different(blue, red). different(blue, green).
```

```
coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).
```

我们有三种颜色。我们告诉Prolog可以在地图着色中使用的不同颜色的集合。接下来，定义一个规则。在这个coloring规则中，告诉Prolog各个州之间的接壤关系，我们完成了。试一下：

```
| ?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).

Alabama = blue
Florida = green
Georgia = red
Mississippi = red
Tennessee = green ?
```


果然，有一种方法可以使用三种颜色对这五个州进行着色。你也可以通过输入a得到另外几种着色组合。通过十几行代码，我们就完成了地图着色。这个逻辑非常简单，即便是小孩子都可以理解。某个时候，你要问自己……

2. 程序在哪

我们没有使用算法！试试选一门过程式编程语言来解决这个问题。你的解决方法容易理解吗？考虑一下如果用Ruby或Io来解决这样复杂的逻辑问题你需要做些什么？一个可能的解决方法如下：

(1) 收集和整理逻辑；

(2) 用程序表达逻辑；

(3) 找出所有可能的解决方法；

(4) 通过程序验证这些可能的解决方法。

你可能不得不一遍又一遍地去编写程序。Prolog让你通过事实和推论来表达逻辑，然后直接提问即可。你不必用这门语言去制作任何具有详细步骤的烹饪食谱。Prolog不是通过编写算法来解决逻辑问题的，而是通过如实地描述真实世界，来呈现计算机可以设法解决的逻辑问题。

让计算机做这些工作吧。

4.2.4　合一，第一部分

现在，到了回过头来介绍一些理论知识的时候了。我们来介绍一下合一（unification）。有些语言使用变量赋值。在Java或Ruby中，举个例子，x = 10意思是将10赋值给变量x。作用在两个结构之间的合一会试图使两个结构完全相同。考虑下面的知识库：

prolog/ohmy.pl

```
cat(lion).
cat(tiger).

dorothy(X, Y, Z) :- X = lion, Y = tiger, Z = bear.
twin_cats(X, Y) :- cat(X), cat(Y).
```

在这个例子中，=意为合一，或者说使等号两侧相同。我们拥有两个事实：lion和tiger都是cat。我们还有两个简单的规则。在规则dorothy/3中，X、Y和Z分别为lion、tiger和bear。在规则twin_cats/2中，X是cat，Y也是cat。我们可以使用这个知识库来解释一下合一。

首先，使用第一个规则。我先编译，然后执行一个不带参数的简单查询。

```
| ?- dorothy(lion, tiger, bear).

yes
```

记住，合一的意思是“找出那些使规则两侧匹配的值”。在右侧，Prolog将X、Y和Z分别绑定为lion、tiger和bear。这些值与左侧相应的值匹配，所以合一是成功的。Prolog报告yes。这个例子非常简单，不过可以为它再增加点趣味。合一在规则的两侧都能工作。

```
| ?- dorothy(One, Two, Three).

One = lion
Three = bear
Two = tiger

yes
```

这个例子多了一个间接层。在子目标中，Prolog使得X、Y和Z分别与lion、tiger和bear合一。在左侧，Prolog使得X、Y和Z分别与One、Two和Three合一，然后报告结果。

现在，让我们转到最后那条规则twin_cats/2。这条规则说如果你能证明X和Y都是cat，那么这条规则就为真。试一下：

```
| ?- twin_cats(One, Two).

One = lion
Two = lion ?
```

Prolog报告了第一种可能。lion和lion都是cat。我们来看一下它是如何得到这个结果的。

(1) 我们发起查询twin_cats(One, Two)。Prolog将One绑定到X，将Two绑定到Y。要处理这个查询，Prolog必须从下述这些目标开始。

(2) 第一个目标是cat(X)。

(3) 我们有两个事实用于匹配，cat(lion)和cat(tiger)。Prolog尝试第一个事实，将X绑定到lion，然后继续下一个目标。

(4) Prolog现在绑定Y到cat(Y)。Prolog使用与处理第一个目标完全相同的办法处理这个目标，选择lion。

(5) 我们已经满足了两个目标，所以这个规则为真。Prolog报告可以让规则为真的One和Two

的值，并且报告yes。

这样，我们有了第一个使规则为真的解决方法。有些时候，一个解决方法就足够了。但有些时候，一个解决方法可能不够用。现在可以使用“;”符号逐个得到其余的解决方法，也可以输入a来获得剩余的所有解决方法。

```
Two = lion ? a

One = lion
Two = tiger

One = tiger
Two = lion

One = tiger
Two = tiger

(1 ms) yes
```

注意，给定目标和相应事实中可用的信息，Prolog将可以列出X和Y所有可能的组合。正如你将在后面看到的那样，合一也能够进行一些基于数据结构的复杂匹配。第一天的学习很充分了。我们将在第二天完成一些更有难度的工作。

4

4.2.5 实际应用中的Prolog

通过这种方式呈现一个“程序”不免让人心生不安。在Prolog中，很少会有一份说明详细步骤的烹饪食谱，你完成时只能看到一份对平底锅中蛋糕的描述。当我学习Prolog时，与一些在实际工作中使用Prolog的人的交谈极大地帮助了我。我曾同Brian Tarbox交谈过，他在一个研究项目中使用这门逻辑语言为海豚研究进行日程安排。

Brian Tarbox访谈录

Bruce：可以谈谈你学习Prolog的经历吗？

Brian：我的Prolog学习要追溯到20世纪80年代末期，那时我就读于马诺阿（Manoa）的夏威夷大学研究生院。我在Kewalo盆地海洋哺乳动物实验室研究宽吻海豚（bottlenosed dolphins）的认知能力。那时，我注意到实验室中的多数讨论都与海豚是如何思考的理论有关。我们主要与一个名为Akeakamai，简称Ake的海豚一起工作。很多讨论都以“嗯，Ake可能看到了这样的情况”作为开始。

我决定在我的硕士论文中尝试建立一个可执行的模型，用于匹配我们对Ake理解世界的能力的判断，或者至少是匹配我们所研究的判断的一个小子集。如果我们的可执行模型可以预测Ake的实际行为，我们将对我们的海豚思考理论更有信心。

Prolog是一门奇妙的语言，但它总是会给出一些非常奇怪的结果。我记得第一次尝试使用Prolog时的情形，我试着写出一行类似x = x + 1的代码。Prolog回应no。其他编程语言一般不会说no，而是会给出一些错误答案或提示编译失败，但是我从来没有见过一门语言回话给我。所

以我致电Prolog的技术支持说：当我尝试修改一个变量值的时候，这门语言对我说“no”。他们问我“为什么想修改一个变量的值？”我的意思是，什么样的语言不允许你修改一个变量的值呢？一旦你深入了解Prolog，你就会理解变量要么具有特定值，要么处于未绑定状态，但在当时我确实是丈二和尚摸不着头脑。

Bruce：你是怎么使用Prolog的？

Brian：我开发了两个主要的系统：海豚模拟器和一个实验室日程安排程序。实验室每天都对四个海豚分别进行四个实验。你必须明白用于研究的海豚是极为有限的资源。每个海豚用于不同的实验并且每个实验需要的人员配置各不相同。一些角色，如海豚驯养师，只可能由少数人承担。其他角色，如数据记录员则可以由很多人完成，不过也需要经过培训。大多数实验需要六到十几名工作人员。我们有研究生、本科生以及地球环境监察志愿者。每个人都有自己的时间安排和各自的技能。找到一个可以利用每个人并且确保所有任务完成的时间安排成为了一名工作人员的专职工作。

我决定尝试实现一个基于Prolog的日程安排生成器。结果证明这简直就是一个为这门语言量身定做的问题。我建立一组事实，描述每个人的技能，每个人的日程安排以及每个实验的需求。之后我就可以简单告诉Prolog“我要这么做”。对于一个实验中列出的每个任务，这门语言都可以找出一个具备所需技能且空闲的人，并将他与那个任务相绑定。它继续向后找，直到要么满足实验的需求，要么无法满足。如果它无法找到一个有效的绑定，它将撤销上一个绑定并再次尝试另外一种组合。最后，它要么找到一个有效的安排，要么报告这个实验限制过多。

Bruce：是否有一些与海豚相关的事实、规则或者断言的有趣例子对我们的读者也很有意义？

Brian：我记得曾经有一次海豚模拟器帮助我们理解了Ake的实际行为。Ake对一套手势标记语言有反应，这套语言包含诸如“钻圈”或“用尾巴扫右边的球”等句子。我们给它发指令，它就会有回应。

我的研究就包括训练海豚理解诸如not这样的新单词。在这种背景下，“touch not ball”意思是除了球之外可以触碰任何东西。对于Ake来说这是一个很难解决的问题，不过这个研究还是顺利地进展了一段时间。不过有一次，无论我们何时给它下指令，它开始都仅是沉入在水下。我们不明白为何会出现这种事情。这是一个非常令人沮丧的情况，因为你无法问一个海豚它为什么要这么做。所以我们把这个训练任务交给了海豚模拟器并且得到了一个有趣的结果。虽然海豚非常聪明，但它们通常会找最简单的答案来回答问题。我们海豚模拟器拥有同样的启发式方法。结果发现针对Ake的手势包含了一个描述池塘中某扇窗户（window）的单词。大多驯养师忘记了这个词，因为它极少使用。海豚模拟器发现了这个规则，即window是对not ball的一个成功的回应。它也是not hoop、not pipe以及not frisbee的成功回应。我们曾多次尝试通过改变池塘中的其他物件组合来防范这种模式，不过很显然我们无法移动窗户。结果证明当Ake沉入池塘底部时，它的位置紧邻窗户，虽然我无法看见窗户。

Bruce：关于Prolog，你最喜欢它哪一点呢？

Brian：声明式的编程模型非常有魅力。一般来说，如果你能把问题描述出来，你就可以解决这个问题。在绝大多数语言中，某些时候我发现自己是在与计算机争论：“你知道我的意思，

就这么做吧。”C和C++编译器错误，诸如“缺少分号”，都是这场争论的表象。如果缺少分号，那就插入一个分号看看是否修正了这个问题。在Prolog中，为了解决日程安排问题，我要做的仅仅是简单地说：“我想要一个像这样的一天，去给我安排吧。”然后Prolog就会去替我安排。

Bruce：什么让你觉得最麻烦？

Brian：Prolog看起来似乎是以一种要么全有要么全无的方式去解决问题，或者至少在我处理的问题上是这样的。在实验室日程安排的问题上，这个系统会花费30分钟处理，然后要么给出一个出色的日程安排，要么简单地打印no。在这里no的意思是我们的限制过度了，没有一个完满的解决方法了。不管怎样，它没有给我们一个不完整的解决方法或者给出任何有关哪里有过度限制的信息。

你在这里看到的是一个极其强大的概念。你无需描述问题的解决方案。你只需描述问题即可。并且这门语言使用逻辑描述问题，只是用纯逻辑。从事实和推论开始，让Prolog完成剩余工作。Prolog程序抽象层次更高。调度和行为模式都是Prolog最为擅长处理的问题。

4.2.6　第一天我们学到了什么

今天，我们学到了Prolog语言的基本构造结构。我们使用纯逻辑对知识进行编码，而不是编写每步操作去指导Prolog形成解决方法。Prolog承担将知识编织起来并查找解决方法的繁重工作。我们将逻辑放入知识库并发起有针对性的查询。

在建立一些知识库后，就可以编译和发起查询了。查询有两种形式，第一种，查询中指定一个事实，Prolog将告诉你这个事实是真还是假。第二种，在查询中使用一个或更多变量，然后Prolog将计算出可以使事实为真的所有可能性。

我们了解到，Prolog执行规则时是按顺序执行规则中的子句。对于每个子句，Prolog都会尝试所有可能的变量组合来满足每个子目标。所有Prolog程序都是这么工作的。

在下一节中，你将看到一些更复杂的推论。我们将在学习中使用数学和更复杂的数据结构，比如列表，同时还要学习迭代列表的策略。

4.2.7　第一天自习

找

- 一些免费的Prolog教程。
- 一个技术论坛（有许多）。
- 一个你正在使用的Prolog版本的在线参考。

做

- 建立一个简单的知识库。描述一些你最喜欢的书籍和其作者。
- 找出知识库中某位作者编写的所有书籍。
- 建立一个描述音乐家和乐器的知识库，同时也描述出音乐家以及他们的音乐风格。
- 找出所有使用吉他的音乐家。

4.3　第二天：离瓦普纳法官开演还有 15 分钟

《雨人》中的主角痴迷于*The People's Court*中的那个脾气暴燥的瓦普纳（Wapner）法官。和多数自闭症患者一样，Raymond痴迷所有熟悉的事物。他缠着要看瓦普纳法官和*The People's Court*。当辛苦地完成这门奇妙语言的学习后，你也许已经做好恍然大悟的准备了。现在，你也许就是那个恍然大悟的幸运读者，不过如果你不是，也请打起精神来。今天，一定有“离瓦普纳法官开演还有15分钟”的时刻。坐好。我们的工具箱里还需要一些工具。你将学到如何使用递归和列表并进行数学运算。让我们开始吧。

4.3.1　递归

Ruby和Io是命令式编程语言。你需要清楚地说明一个算法的每一步。Prolog是我们看到的第一门声明式编程语言。当你处理事物集合时，如列表或树，你会经常使用递归而不是迭代。我们将学习递归并使用它解决一些包含基本推论的问题。接下来我们会将相同的技术运用到列表和数学运算上去。

看一看下面的数据库。它表示了沃尔顿家族的全部家谱，他们都是1963年的一部电影及其后续系列影片中的角色。这个库表示了一种父子关系以及由此推断出的祖先关系。由于一个祖先可能是一个父亲、祖父或曾祖父，所以需要嵌套使用规则或使用迭代。由于使用的是一门声明式语言，因此将使用嵌套规则。在`ancestor`子句中的一个子句会使用`ancestor`子句。在这个例子中，`ancestor(Z, Y)`是一个递归的子目标。下面是这个知识库：

prolog/family.pl

```
father(zeb,         john_boy_sr).
father(john_boy_sr, john_boy_jr).

ancestor(X, Y) :-
    father(X, Y).
ancestor(X, Y) :-
    father(X, Z), ancestor(Z, Y).
```

`father`是实现递归子目标的核心事实。规则`ancestor/2`有两个子句。如果一个规则由多个子句组成，那么其中一个子句为真，则这个规则为真。把子句间的逗号看成是条件“与”的关系，把子句之间的句号看成是条件“或”的关系。第一个子句表明“如果X是Y的`father`，那么X是Y的`ancestor`”。这是一个直接关系。我们可以测试一下这个规则：

```
| ?- ancestor(john_boy_sr, john_boy_jr).

true ?

no
```

Prolog报告为真，`john_boy_sr`是`john_boy_jr`的`ancestor`。第一个子句取决于一个事实。第二个子句更为复杂：`ancestor(X, Y) :- father(X, Z), ancestor(Z, Y)`。这个子句

表明如果我们可以证明X是Z的father并且同一个Z是Y的一个ancestor，那么X就是Y的一个ancestor。

我们测试第二个子句：

```
| ?- ancestor(zeb, john_boy_jr).

true ?
```

是的，zeb是john_boy_jr的一个ancestor。和以往一样，我们可以在查询中使用变量：

```
| ?- ancestor(zeb, Who).

Who = john_boy_sr ? a

Who = john_boy_jr

no
```

我们看到zeb是john_boy_jr和john_boy_sr的一个共同ancestor。这个ancestor谓词也可以反过来用：

```
| ?- ancestor(Who, john_boy_jr).

Who = john_boy_sr ? a

Who = zeb

(1 ms) no
```

这太美妙了，我们可以在知识库中使用这个规则实现两个目的：寻找祖先和后代。

下面是一段简短的警告。当你使用递归子目标时，你需要小心。因为每个递归的子目标都会使用栈空间，最终你很可能会耗尽栈空间。声明式语言通常使用一种称为*尾递归优化*（tail recursion optimization）的技术来解决这个问题。如果你将一个递归的子目标放到递归规则的末尾，Prolog会通过丢弃调用栈来优化这次调用，并保持内存占用不变。这里的调用就是一个尾递归，因为递归子目标ancestor(Z, Y)是递归规则中的最后一个目标。如果你的Prolog程序因耗尽栈空间而崩溃的话，那么就应该知道是时候使用尾递归对程序进行优化了。

最后那点整理工作已经解决了，让我们开始看看列表和元组。

4.3.2　列表和元组

列表和元组是Prolog的重要组成部分。你可以像[1, 2, 3]这样指定一个列表，像(1, 2, 3)这样指定一个元组。列表是变长容器，而元组则是定长容器。当你从合一的角度考虑时，列表和元组都会变得更加强大。

合一，第二部分

记住，当Prolog对变量进行合一操作时，它会尝试使左右两侧相匹配。如果两个元组拥有的元素数量相同且每个元素可以合一，则它们就可以匹配。看看下面几个例子。

```
| ?- (1, 2, 3) = (1, 2, 3).

yes
| ?- (1, 2, 3) = (1, 2, 3, 4).

no
| ?- (1, 2, 3) = (3, 2, 1).

no
```

如果两个元组的所有元素可以合一，则这两个元组可以合一。第一对元组精确地匹配，第二对元组由于拥有的元素数量不同而无法匹配，第三对元组则因元素顺序不同而无法匹配。接下来加入一些变量：

```
| ?- (A, B, C) = (1, 2, 3).

A = 1
B = 2
C = 3

yes
| ?- (1, 2, 3) = (A, B, C).

A = 1
B = 2
C = 3

yes
| ?- (A, 2, C) = (1, B, 3).

A = 1
B = 2
C = 3

yes
```

变量在哪一侧没有关系。如果Prolog认为它们相同，它们就可以合一。现在，看看一些列表，它们与元组的用法相同：

```
| ?- [1, 2, 3] = [1, 2, 3].

yes
| ?- [1, 2, 3] = [X, Y, Z].

X = 1
Y = 2
Z = 3

yes
| ?- [2, 2, 3] = [X, X, Z].
```

```
X = 2
Z = 3

yes
| ?- [1, 2, 3] = [X, X, Z].

no
| ?- [] = [].
```

最后两个例子很有趣。[X, X, Z]和[2, 2, 3]可以合一，因为Prolog设置X = 2后可以使它们相同。[1, 2, 3] = [X, X, Z]不能合一，因为X同时用在第一个和第二个元素位置上，但两个元素的值不同。列表拥有一项元组所不具备的能力，即你可以通过[Head|Tail]解构列表。当你将一个列表与这种结构合一时，Head将绑定列表的第一个元素，而Tail将绑定剩余元素，像这样：

```
| ?- [a, b, c] = [Head|Tail].

Head = a
Tail = [b,c]

yes
```

[Head|Tail]不能与一个空列表合一，不过单元素列表可以。

```
| ?- [] = [Head|Tail].

no
| ?- [a] = [Head|Tail].

Head = a
Tail = []

yes
```

你可以使用各种复杂的组合：

```
| ?- [a, b, c] = [a|Tail].

Tail = [b,c]

(1 ms) yes
```

Prolog匹配a，并将剩余部分与Tail合一。或者可以将这个Tail再拆分成Head和Tail：

```
| ?- [a, b, c] = [a|[Head|Tail]].

Head = b
Tail = [c]

yes
```

或抓取第三个元素：

4

```
| ?- [a, b, c, d, e] = [_, _|[Head|_]].

Head = c

yes
```

“_”是一个通配符，它可以与任何对象合一。大体上它的含义是：“我不关心这个位置上的元素是什么。”我们告诉Prolog略过前两个元素，将剩余元素拆分为Head和Tail。Head将抓取第三个元素，末尾的_将抓取Tail，并忽略列表的其余元素。

掌握上述内容后，你就可以开始工作了。合一是一个功能强大的工具。将它与列表和元组结合在一起会使它变得更为强大。

现在，你应该基本掌握了Prolog中的核心数据结构以及合一的工作方式。现在我们准备将这些与规则和断言结合在一起，使用逻辑解决一些基本数学运算问题。

4.3.3　列表与数学运算

在接下来的例子中，我会向你展示在列表上使用递归和数学运算。这些例子包括计数、汇总和求平均值。这里用五个规则来完成所有这些困难的工作。

prolog/list_math.pl

```
count(0, []).
count(Count, [Head|Tail]) :- count(TailCount, Tail), Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.

average(Average, List) :- sum(Sum, List), count(Count, List), Average is Sum/Count.
```

最简单的例子是count，可以像下面这样使用它：

```
| ?- count(What, [1]).

What = 1 ? ;

no
```

这些规则很简单。对一个空列表计数结果为0。一个非空列表的计数等于对列表Tail的计数加上1。下面逐步说明它的工作方式。

- 发起查询count(What, [1])，由于列表非空，无法与第一个规则合一。我们继续满足第二个规则count(Count, [Head|Tail])中的目标。进行合一操作，What绑定为Count，Head绑定为1，Tail绑定为[]。
- 合一后，第一目标变为count(TailCount, [])。我们尝试证明这个子目标。这次，我们与第一条规则进行合一。TailCount绑定为0。现在第一个规则满足了，所以接下来处理第二个目标。
- 现在，对Count的求值为TailCount + 1。我们可以合一变量。TailCount绑定为0，因此将Count绑定为0+1或1。

就是这样。我们没有定义递归过程，只是定义了逻辑规则。下一个例子是对列表中的元素求和。下面是这些规则的源代码：

```
sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.
```

这段代码与count规则的工作方式十分类似。它也包含两个子句，一个基本情况和一个递归情况。用法是相似的：

```
| ?- sum(What, [1, 2, 3]).

What = 6 ? ;

no
```

如果你用命令式的视角看待它的话，sum工作方式确实满足了你对递归语言的期望。空列表的总和是0，其余元素的总和是列表的Head元素加上Tail的总和。

但是这里还有另外一个解释。我们并没有真正告诉Prolog如何计算总和。我们只是将sum描述为规则和一些目标。为了满足其中部分目标，逻辑引擎必须满足一些子目标。声明式的解释是这样的："一个空列表的总和是0，如果可以证明列表Tail的总和加上Head是Total，那么列表的总和就是Total。"我们用证明目标和子目标的想法替换了递归过程。

同样，空列表的计数为0。非空列表的计数为列表的Head加上列表Tail的计数值。

如同逻辑一样，这些规则可以彼此依赖。例如，你可以将sum和count一起用于计算平均值。

```
average(Average, List) :- sum(Sum, List), count(Count, List), Average is Sum/Count.
```

这样，List的平均值为Average，如果你可以证明：

- List的总和是Sum
- List的计数是Count，并且
- Average是Sum/Count

它的工作方式和你预计的一样：

```
| ?- average(What, [1, 2, 3]).

What = 2.0 ? ;

no
```

4.3.4　在两个方向上使用规则

说到这里，你应该很好地理解了递归是如何工作的了。我将改变一些节奏，讨论一个被称为append的规则。如果List3为List1+List2，那么规则append(List1, List2, List3)为真。这是一个功能强大的规则，你可以通过多种方式使用它。

这些简短的代码放在一起将构成强大的功能，你可以通过多种不同方式使用它们。下面是一个测谎器：

```
| ?- append([oil], [water], [oil, water]).

yes
| ?- append([oil], [water], [oil, slick]).

no
```

下面是一个列表构造器：

```
| ?- append([tiny], [bubbles], What).

What = [tiny,bubbles]

yes
```

下面的代码用于列表减法操作：

```
| ?- append([dessert_topping], Who, [dessert_topping, floor_wax]).

Who = [floor_wax]

yes
```

下面的代码用于计算出可能的排列：

```
| ?- append(One, Two, [apples, oranges, bananas]).

One = []
Two = [apples,oranges,bananas] ? a

One = [apples]
Two = [oranges,bananas]

One = [apples,oranges]
Two = [bananas]

One = [apples,oranges,bananas]
Two = []

(1 ms) no
```

一个规则给了你四种能力。你也许在想，构造这样一个规则可能需要用很多代码。接下来看看究竟需要多少。重新实现Prolog的`append`，不过将它称之为`concatenate`。我们将通过如下几个步骤完成。

(1) 编写一个规则`concatenate(List1, List2, List3)`，它可以将一个空列表与`List1`连接在一起。

(2) 添加一个规则，它可以将`List1`中的一个元素与`List2`连接在一起。

(3) 添加一个规则，它可以将`List1`中的两个元素或三个元素与`List2`连接在一起。

(4) 看看我们可以泛化哪些东西。

让我们开始吧。第一步是将一个空列表与`List1`连接在一起。这是一个极其容易编写的规则：

prolog/concat_step_1.pl

```
concatenate([], List, List).
```

没问题。如果第一个参数是一个列表，并且后两个参数相同，则concatenate为真。

它可以正常工作：

```
| ?- concatenate([], [harry], What).

What = [harry]

yes
```

下一步。我们添加一个规则，将List1的第一个元素连接到List2的前面。

prolog/concat_step_2.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
```

对于concatenate(List1, List2, List3)，我们将List1分成Head和Tail，其中Tail是一个空列表。我们将第三个元素分成Head和Tail，将List1的Head和List2放在一起作为Tail。记得编译知识库。它工作得很好：

```
| ?- concatenate([malfoy], [potter], What).

What = [malfoy,potter]

yes
```

现在，可以定义另外两个规则以连接长度为2和3的列表。它们以同样的方式工作：

prolog/concat_step_3.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
concatenate([Head1|[Head2|[]]], List, [Head1, Head2|List]).
concatenate([Head1|[Head2|[Head3|[]]]], List, [Head1, Head2, Head3|List]).

| ?- concatenate([malfoy, granger], [potter], What).

What = [malfoy,granger,potter]

yes
```

我们这里看到的只是一个基本情况和基本策略，即每个子目标缩小第一个列表，增加第三个列表。第二个列表保持不变。现在我们有足够信息去泛化一个结果。下面是使用了嵌套规则的concatenate：

prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
  concatenate(Tail1, List, Tail2).
```

这个简洁的代码块有个令人难以置信的简单解释。第一个子句表明将一个空列表与List连接在一起，而你将得到那个List。第二个子句表明如果List1和List3的Head相同，并且你可以证明将List1的Tail和List2连接在一起将得到List3的Tail，那么将List1和List2连接在一起将会得到List3。这个解决方法的简单和优雅是Prolog强大功能的实际证明。

接下来看看针对查询concatenate([1, 2], [3], What)它将做什么。我们将执行每一步的合一操作。记住这里使用的是嵌套规则，所以每次设法证明一个子目标时，都会有一份不同的变量副本。我将用一个字母标记出重要的变量，以便于你理解这个规则。当Prolog尝试去证明下一个子目标时，我都会向你展示发生了什么。

- 用这个开始：

  ```
  concatenate([1,2], [3], What)
  ```

- 第一个规则没有使用，因为[1, 2]不是一个空列表。执行合一操作后得到下面这个结果：

  ```
  concatenate([1|[2]], [3], [1]Tail2-A]:-concatenate([2], [3], Tail2-A]
  ```

  除了第二个Tail外，所有都合一了。我们现在处理这个目标。对右侧进行合一操作。

- 我们尝试应用规则concatenate([2], [3], [Tail2-A])。我们会得到这样的结果：

  ```
  concatenate([2|[ ]], [3], [2|Tail2-B]) :- concatenate([ ], [3], Tail2-B)
  ```

  注意，Tail2-B是Tail2-A的Tail，它与原先的Tail2不同。不过现在，我们不得不再次对右侧进行合一操作。

- concatenate([ ], [3], Tail2-C) :- concatenate([ ], [3], [3])。
- 这样，我们知道Tail2-C是[3]。现在我们可以沿着这条链回溯。Tail2-C是[3]，意味着[2|Tail2-B]是[2, 3]，最后[1|Tail2]是[1,2,3]。What是[1,2,3]。

这里Prolog为你做了大量工作。重温一下这个列表，直到你真正理解它。对嵌套的子目标进行合一操作是解决本书中一些高级问题的核心概念。

现在，你已经见识到了Prolog中功能最丰富的函数之一。花点时间仔细研究一下这些方法吧，并确保你真正理解了它们。

4.3.5　第二天我们学到了什么

在这一节中，我们学习了Prolog用于组织数据的基本组成部分：列表和元组。我们还使用了嵌套规则，它可以表达一些在其他语言中使用迭代处理的问题。我们深入了解了Prolog的合一以及Prolog是如何使得:-或=的两侧相匹配的。我们看到，当编写规则时，描述的是逻辑规则而不是算法，并且让Prolog按照自己的方式得到解决方法。

我们也使用了数学运算。我们学会了使用基本算术和嵌套子目标计算总和和平均值。

最后，我们学会了使用列表。我们将一个列表中的一个或多个变量与一些变量匹配，不过更重要的是，使用[Head|Tail]模式匹配将列表的Head和剩余元素与变量相匹配。我们使用这种技术递归地迭代整个列表。这些构建组件将成为第三天中解决复杂问题的基础。

4.3.6 第二天自习

找

- 一些斐波那契数列和阶乘的实现。它们是怎么工作的？
- 使用Prolog解决现实问题的社区。如今人们使用Prolog解决什么问题？

如果你正在找一些你可以潜心去做的更高级任务，试试下面这些问题。

- 实现一个汉诺塔问题。它是如何工作的？
- 有哪些处理not表达式的问题？为什么在使用Prolog时需要小心地对待否定？

做

- 翻转一个列表中的元素次序。
- 找出列表中最小的元素。
- 对一个列表中的元素进行排序。

4.4 第三天：维加斯的爆发

你应该更能理解我为什么为Prolog选择“雨人”，那个自闭的博学的人了。虽然这有时很难理解，但用这种方式思考编程确实令人吃惊。《雨人》中我最喜欢的是当Raymond的弟弟意识到Raymond可以算牌的那一幕。Raymond和他的弟弟去了拉斯维加斯并在赌城里面大赚特赚。在这一节中，你将看到Prolog让你开心的一面。本节中的例子编码既令人恼怒，也令人兴奋。我们将解决两个著名的难题，而这恰好是Prolog的强项，解决资源受限系统的问题。

你可能想自己试试解决其中的一些难题。如果你确定这么做，那就去尝试为每个你了解的游戏描述规则，而不是告知Prolog一个步骤详细的解决方法。我们从一个小小的数独问题开始，然后在日常练习中给你出一个更复杂的数独问题。最后，我们解决经典的八皇后问题。

4.4.1 解决数独问题

我对编程解决数独问题很着迷。数独是一个网格，其中有行、列和格子。典型的数独问题是一个9 × 9的网格，其中有一些格子已填充了数字，还有一些是空白的。9 × 9的方形网格中的每个单元都应有一个数字，从1 ~ 9。你的任务就是填充这个网格，使得每行、每列里的每个格子上的数字都只出现一次。

我们用一个4 × 4的数独问题开始。虽然其解决方法短小些，但思路是一样的。正如我们知道的那样，让我们开始描述这个世界吧。抽象地说，我们有一个题板，上面有4行、4列16个格子。下表展现了这些格子：

```
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
```

第一个任务是决定查询是什么样子的，这非常简单。有一个难题及一个解决方法，用`sudoku(Puzzle, Solution)`的形式表示。使用者用一个列表表示这个难题，用下划线代替未知数字，像下面这样：

```
sudoku([_, _, 2, 3,
        _, _, _, _,
        _, _, _, _,
        3, 4, _, _],
        Solution).
```

如果存在一个解决方法，Prolog将提供这个解决方法。如果我们用Ruby来解决这个难题的话，那我们将为解决这个难题的算法苦恼。但是使用Prolog，就不必这样。我们只需要提供这个游戏的规则即可。下面就是游戏规则。

- ❑ 对于一个已经解决了的难题，难题中的数字与解决方法中的数字应该是相同的。
- ❑ 数独的题板是一个有着16个单元的网格，填充值从1～4。
- ❑ 题板有 4 行、4 列以及 4 ×4个格子。
- ❑ 如果每行、每列里的每个格子都没有重复数字，那么这个难题就被解决了。

让我们从头开始。解决方法和难题中的数字应该匹配：

prolog/sudoku4_step_1.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle.
```

我们实际上已经有了一些进展。我们的“数独解决者”可以处理没有空白单元的情况：

```
| ?- sudoku([4, 1, 2, 3,
             2, 3, 4, 1,
             1, 2, 3, 4,
             3, 4, 1, 2], Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

输出格式不是很美观，不过意图非常清晰。我们得到了逐行排列的16个数字。不过我们似乎还不满足：

```
| ?- sudoku([1, 2, 3], Solution).

Solution = [1,2,3]

yes
```

现在，这个题板是无效的，不过数独解决者却报告有一个有效的方法。显然，应该给题板加上16个元素的限制。我们还有另外一个问题，单元格里可以填写任何值：

```
| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

Solution = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]

yes
```

对于一个有效的解决方法，它填充的数字应该在1～4的范围内。这个问题会从两个方面影响我们。第一，这会产生一些无效的解决方法。第二，Prolog没有足够的信息，无法检测出每个单元格的合法值。换句话说，结果集是不稳定的。这意味着没有设定规则约束每个单元格中的合法值，所以Prolog无法猜测这些值是什么。

接下来为游戏添加下一条规则来解决这些问题。规则2表明一个题板有16个单元，每个单元中数字的范围为1～4。GNU Prolog使用内置的被称作`fd_domain(List, LowerBound, UpperBound)`的谓词（predicate）来表达合法值。如果列表中所有元素都在`LowerBound`和`UpperBound`之间，包括`UpperBound`，那么这个谓词为真。我们只需要保证`Puzzle`中的所有数字值在1～4之间。

prolog/sudoku4_step_2.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle,
        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],
        fd_domain(Puzzle, 1, 4).
```

我们将`Puzzle`与一个拥有16个变量的列表进行合一，并限制单元格的值域为1～4。现在如果`Puzzle`无效，我们将得到失败信息。

```
| ?- sudoku([1, 2, 3], Solution).

no


| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

no
```

现在，开始实现解决方法的主要部分。规则3表示一个由行、列和格子组成的题板。我们将难题分隔成行、列和格子。现在，你将看到我们为何这样命名每个单元格了。这种描述行的方法很直观：

```
Row1 = [S11, S12, S13, S14],
Row2 = [S21, S22, S23, S24],
Row3 = [S31, S32, S33, S34],
Row4 = [S41, S42, S43, S44],
```

对列的描述也是一样的：

```
Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],
```

还有格子：

```
Square1 = [S11, S12, S21, S22],
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44].
```

现在，我们已经将题板分成了几块，可以继续下一条规则了。如果所有行、列和格子都没有重复的元素，那么题板就是有效的。我们将使用一个GNU Prolog谓词做重复元素的检查。如果所有在`List`中的元素都不同，那么`fd_all_different(List)`返回真。我们需要构建一个规则来检测所有行、列和格子都是有效的。我们使用下面这个简单的规则来完成这个任务：

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).
```

如果其中所有的列表都是不同的，那么这个谓词就是有效的。第一句表达的是一个空列表是有效的。第二句表达的是，如果第一个元素列表的各项都不同并且剩余列表都是有效的，那么这个列表就是有效的。

剩下的工作就是调用`value(List)`规则了：

```
valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

不管你是否相信，我们完成了。这个方法可以解决一个4 × 4的数独问题：

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

用一个更友好的形式输出结果，如下所示：

```
4 1 2 3
2 3 4 1
1 2 3 4
3 4 1 2
```

下面是全部的程序代码：

prolog/sudoku4.pl

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).
```

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle,

        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],

        fd_domain(Solution, 1, 4),

        Row1 = [S11, S12, S13, S14],
        Row2 = [S21, S22, S23, S24],
        Row3 = [S31, S32, S33, S34],
        Row4 = [S41, S42, S43, S44],

        Col1 = [S11, S21, S31, S41],
        Col2 = [S12, S22, S32, S42],
        Col3 = [S13, S23, S33, S43],
        Col4 = [S14, S24, S34, S44],

        Square1 = [S11, S12, S21, S22],
        Square2 = [S13, S14, S23, S24],
        Square3 = [S31, S32, S41, S42],
        Square4 = [S33, S34, S43, S44],

        valid([Row1, Row2, Row3, Row4,
               Col1, Col2, Col3, Col4,
               Square1, Square2, Square3, Square4]).
```

如果你未曾体会到Prolog的乐趣，那么这个例子应该可以让你体会到了。程序在哪里？我们没有编写什么程序。我们只是描述了这个游戏的规则：一个有着16个单元格的题板，每个单元格中数字值的范围是1～4，并且没有任何行、列里的格子中有重复的值。这个问题花了几十行代码就解决了，并且没有用到任何数独解决策略方面的知识。在4.4.4节中，你将有机会解决一个有九格子的数独问题，也不会太困难。

这是很好的例子，Prolog很擅长解决此类问题。我们有一组约束，它们易于表达但却难于解决。让我们看看另一个涉及资源严重受限的难题：八皇后问题（Eight Queens）。

4.4.2 八皇后问题

要解决八皇后问题，你需要将八个皇后放在一个棋盘上。每一行、列以及对角线上只能有一个皇后。也许它从表面上看起来是一个微不足道的问题，更像是一个孩子们的游戏。但是在另外一个层面上，你可以将这些行、列和对角线看做受限的资源。我们的行业中充满了解决资源受限系统的难题。让我们看看如何用Prolog解决这个问题吧。

首先，看看查询应该是什么样子的。我们将每个皇后表示为`(Row, Col)`，一个包含行和列号的元组。棋盘是一个元组的列表。如果我们有一个合法的棋盘，`eight_queens(Board)`会返

回成功。我们的查询像下面这样：

```
eight_queens([(1, 1), (3, 2), ...]).
```

下面看看解决这个问题需要满足的目标吧。如果你想亲自解决这个游戏问题而不想借助这里的解决方法，那就看看这些目标吧。我在本章末尾才会展示全部方法。

- 一个棋盘上有八个皇后。
- 每个皇后有一个行号和一个列号，行号和列号取值范围都是1 ~ 8。
- 任意两个皇后不可以共享一行。
- 任意两个皇后不可以共享一列。
- 任意两个皇后不可以共享一个对角线（西南到东北）。
- 任意两个皇后不可以共享一个对角线（西北到东南）。

行与列必须是唯一的，不过我们必须更加小心地处理对角线。每个皇后都在两条对角线上，一条从左下角（西南）到右上角（东北），另一条从左上角到右下角，如图4-2所示。不过针对这些规则编码应该相对容易。

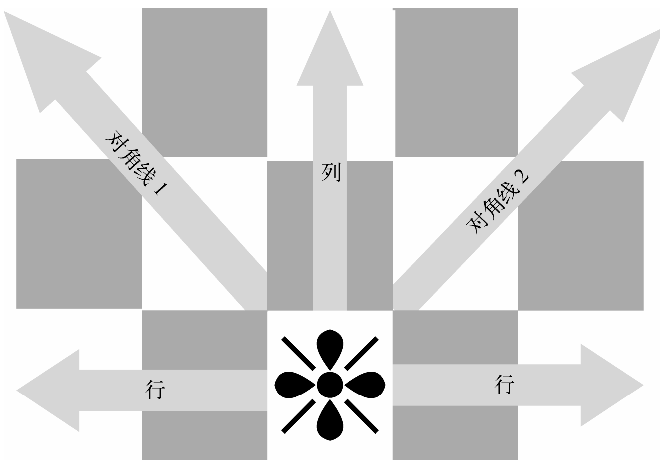


图4-2 八皇后规则

我们还是从第一个目标开始。一个棋盘有八个皇后。这意味着我们的列表大小必须是8。这很容易做。我们可以使用前面介绍过的count谓词，或者我们可以简单地使用Prolog内置的谓词length。如果List有N个元素，那么length(List, N)将返回真。这次，我将带你一起完成解决整个问题所需要的每个目标，而不是用例子说明它们。这里是第一个目标：

```
eight_queens(List) :- length(List, 8).
```

接下来，需要保证列表中的每个皇后都是有效的。我们编写一个规则用于检查皇后是否有效。

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).
```

谓词member刚好用于完成你所想的事情，它用来检查成员资格。如果一个皇后的行与列都是1～8范围内的数字，那么这个皇后就是有效的。接下来，我们将编写一个规则检查整个棋盘是否是由有效的皇后组成的。

```
valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).
```

一个空棋盘是有效的。如果一个非空棋盘的第一个元素是一个有效的皇后并且其余的皇后也是有效的，那么这个棋盘就是有效的。

我们继续。下一个规则是两个皇后不能共享同一行。为了解决后续几个约束，我们需要一点帮助。先将程序分成几片，这样可以帮助我们描述问题：什么是行、列和对角线？首先是行。我们编写一个名为rows(Queens, Rows)的函数。如果Rows是由所有皇后的Row元素组成的列表，那么这个函数应该为真。

```
rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).
```

4

这里需要一点想象力，但不多。对一个空Queens列表执行rows的结果是一个空Rows列表。如果Queens列表中第一个元素的Row与Rows列表中的第一个元素相匹配，并且如果对Queens的Tail列表执行rows的结果是Rows的Tail列表，那么rows(Queens, Rows)的结果就是Rows列表。如果你对此仍感到困惑，可以用测试数据做一些测试来帮助理解。幸运的是，列也采用了相同的工作方式，我们将用列替代行：

```
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).
```

这个逻辑与rows恰好相同，不过这回匹配的是皇后元组的第二个元素，而不是第一个元素。

我们继续给对角线编号。最容易的编号方法就是做一些简单的加减法。如果北和西是1，给左上角（西北）到右下角（东南）的对角线赋一个值Col-Row。下面是一个用于抓取对角线元素的谓词：

```
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

这个规则工作原理恰好类似rows和cols，不过我们还有一个限制：Diagonal is Col - Row。注意，这不是一个合一！它是一个is谓词，它将为这个解决方法打下良好基础。最后，我们会抓取从右下角（东南）到左上角（西北）的对角线元素，如下所示：

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).
```

这个公式用了些小技巧，不过你可以测试几个值，直到你确信行列编号之和相同的皇后真的在同一条对角线上。既然我们已经有了帮我们描述行、列和对角线的规则，剩下要做的就是确保行、列和对角线列表中的元素都是不同的。

下面是这个解决方法的全部代码。最后8个子句是对rows和columns的测试。

prolog/queens.pl

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  length(Board, 8),
  valid_board(Board),

  rows(Board, Rows),
  cols(Board, Cols),
  diags1(Board, Diags1),
  diags2(Board, Diags2),

  fd_all_different(Rows),
  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

现在，你可以运行这个程序了。它将马不停蹄地一直运行。有太多的组合需要高效地整理了。想想看，我们的条件是每一行上有且只有一个皇后。我们可以提供一个像下面这样的题板，并迅速启动这个解决方法：

```
| ?- eight_queens([(1, A), (2, B), (3, C), (4, D), (5, E), (6, F), (7, G), (8, H)]).

A = 1
B = 5
C = 8
D = 6
E = 3
F = 7
G = 2
H = 4 ?
```

这个程序工作得很好，不过它工作得过于“辛苦”。我们可以很容易地删除行选择并简化API。下面是一个稍作优化的版本：

prolog/optimized_queens.pl

```
valid_queen((Row, Col)) :- member(Col, [1,2,3,4,5,6,7,8]).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  Board = [(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)],
  valid_board(Board),

  cols(Board, Cols),
  diags1(Board, Diags1),
  diags2(Board, Diags2),

  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

逻辑上，我们已经做出了一个重大的修改。用(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)来匹配Board，以显著地减少中间结果交换的次数，同时删除了所有与行有关的规则。结果显示了出来。在我的“老古董”MacBook上，所有解决方法都会在3分钟内计算完毕。

最终的结果再次令人满意。我们对解决方法的知识了解有限。我们只是描述了游戏的规则，并且应用了一些逻辑提高了问题的解决效率。只要遇到合适的问题，我发现我真的会融入Prolog的世界中。

4.4.3　第三天我们学到了什么

今天，我们将Prolog中使用的一些想法放在一起解决一些经典的难题。这些基于约束的问题具有许多与经典的行业应用相同的特性。列表约束，以及快速创建解决方法。我们永远不会去想以命令式的方式进行一次9个表的SQL连接操作，然而对于以这种方式解决逻辑问题，我们根本不会感到丝毫奇怪 。

我们以一个数独问题开始。Prolog的解决方法十分简单。将16个变量映射到行、列和格子上。然后，我们描述游戏的规则，迫使每行、每列里的每个格子上的变量都是独一无二的。然后，Prolog有条不紊地检查所有可能性，并快速获得一个解决方法。我们使用通配符和变量构建了一个直观的API，不过并没有提供任何有关解决方法技术方面的帮助。

接下来，使用Prolog去解决八皇后问题。再次对游戏规则编码并且让Prolog得出解决方法。这个经典问题是计算密集型的，共有92种可能的解决方法，不过即使是这个简单的方法，也可以在很短的时间内将问题解决。

我仍然不知道用于解决高级数独问题的所有诀窍和技术，但是使用Prolog我就不需要知道这些。我只需要知道如何玩这种游戏即可。

4.4.4　第三天自习

找

- Prolog也有输入/输出功能。找出可以打印输出变量的`print`谓词。
- 找到一种通过`print`谓词仅输出成功的解决方法的方式。它是如何工作的?

做

- 修改数独解决器用来解决6 × 6数独（每个格子是3 × 2）和9 × 9数独问题。
- 让数独解决器输出格式更美观的解决方法。

如果你是一个解题爱好者，你可能会迷失在Prolog中。如果你想深入研究我曾展示的难题，八皇后问题是一个好的起点。

- 采用一个皇后列表的方式解决八皇后问题。使用一个1 ~ 8范围内的数字代表每个皇后，而不是用元组。通过皇后在列表中的位置获取其行号并且通过其在列表中的值获得其列号。

4.5　趁热打铁

Prolog是这本书中较老的语言之一，不过其思想在今天依旧有其吸引力和价值。Prolog的含义是用逻辑编程。我们使用Prolog去处理由子句组成的规则，而子句又是由一系列的目标组成的。

Prolog编程有两个主要步骤。开始是构建一个由逻辑事实和推论组成的关于问题域的知识库。接下来，编译知识库，并针对问题域进行提问。一些问题可以是断言式的，Prolog会用yes或no来回应。其他查询带有变量。Prolog会填充这些空白使得这些查询为真。

Prolog没有使用简单的变量赋值，而是使用一种被称为合一的过程，该过程使得一个系统两侧的变量相匹配。有时，Prolog对一个推论进行变量合一时会尝试多种可能的变量组合。

4.5.1 核心优势

Prolog适用类型广泛的问题，从航空调度到金融衍生产品。Prolog有着一条不轻松的学习曲线，不过Prolog解决的那些苛刻问题往往会让这门语言或者其他类似的语言物有所值。

回想一下Brian Tarbox关于海豚的研究工作。他能够作出关于这个世界的简单推论，然后能够通过一个有关海豚行为的复杂推论作出了突破性的进展。他还能利用极为有限的资源，使用Prolog找出适合的日程安排。下面是当前一些使用Prolog的活跃领域。

1. 自然语言处理

也许Prolog是第一种用于进行语言识别的语言。特别是，Prolog语言模型可以采用自然语言，应用基于事实和推论的知识库，并且可以用具体的适于计算机的规则表达那些复杂的不精确的语言。

2. 游戏

游戏变得越来越复杂，特别是对竞赛者或敌人行为的建模。Prolog模型能够很轻松地表达系统中其他角色的行为。Prolog也能为不同类型的敌人构建不同的行为，使得用户体验更加逼真愉快。

3. 语义网

语义网是为网络上的服务与信息提供附加含义的一种尝试，从而更容易满足大家的需求。资源描述语言（resource description language, RDF）提供对资源的一个基本的描述。服务器可以将这些资源编译为一个知识库。这些知识，再加上Prolog的自然语言处理就能提供丰富的用户体验。Web服务器上提供了许多此类功能的Prolog包。

4. 人工智能

人工智能（AI）关注让机器拥有智能。智能可能表现为多种形式，不过在每种情况下，一些“代理”会基于复杂规则对行为进行修改。Prolog在这个领域很擅长，尤其是当这些规则是明确的且基于正式逻辑的。为此，Prolog有时被称为一门*逻辑编程语言*。

5. 调度

Prolog擅长处理有限资源。许多厂商使用Prolog实现操作系统调度器以及其他高级的调度器。

4.5.2 不足之处

Prolog经受住了时间的考验。不过这门语言在许多方面仍然过时了，并且它确实有一些明显的局限。

1. 功用

Prolog擅长其核心领域，它专注的目标是逻辑编程。它不是一门通用的编程语言，它也有一些有关语言设计方面的限制。

2. 超大数据集合

Prolog使用了一个深度优先搜索的决策树，它使用所有可能组合与规则集合相匹配，并且其编译器对这个过程做了很好的优化。不过，这个策略需要进行大量计算，特别是当数据集规模非常大的时候。这也迫使Prolog用户必须理解语言的工作原理以保持数据集的规模在可控范围内。

3. 混合命令式和声明式模型

和许多函数式语言一样，特别是那些严重依赖递归的语言，你必须理解Prolog是如何解决递归规则的。你必须经常使用尾递归规则去完成中等规模的问题。构建一个基于小数据集但无法扩展的Prolog应用相对容易，但必须深入理解Prolog的工作原理才能更有效地设计出在可接受的层次上进行扩展的规则。

4.5.3　最后思考

当我学习完这本书中的各门编程语言时，我经常自责，感觉这些年来一直都在杀鸡用牛刀。Prolog就是我在不断学习的过程中的一个特别深刻的例子。如果你发现一个特别适合Prolog解决的问题，那就利用Prolog解决吧。只有这样，你才能更好地将这门基于规则的语言与其他通用语言结合在一起使用。就像你在Ruby或Java中使用SQL一样。如果能很好地将它们结合在一起，你很可能最终因此脱颖而出。

第 5 章

Scala

5

我们不是绵羊。

——剪刀手 Edward

到目前为止，我已经介绍过三种编程语言以及三种不同的编程范型（programming paradigm）了。Scala是第四种。它是一种混合编程语言，混合编程语言在两种不同编程范型之间搭建一座桥梁以弥合差异。在这里，这座桥梁搭建在面向对象语言（如Java）以及函数式语言（如Haskell）之间。从这个意义上讲，Scala可以说是一个科学怪人，但却不是一个怪物。想一想《剪刀手爱德华》[①]这部电影。

在Tim Burton的这部超现实影片中，Edward是一个拥有一双剪刀手的机器人男孩儿，同时他也是一直以来我最喜欢的银幕角色之一。在这部美丽的影片中，Edward是一个很吸引人的角色。他常常笨手笨脚，不过有时也令人吃惊，并总是表现出一副独特的表情。有时，他能用他的剪刀手做出一些不可思议的事情，有时又因笨拙而被人羞辱。他经常因为其标新立异的行为而被误解，甚至被指责为“迷失正道”。然而在一次变得更坚强的时刻，这个害羞的男孩却说出了“我们不是绵羊”。他说得没错。

5.1　关于 Scala

随着对计算机程序的需求越来越复杂，计算机语言也在发展演化。每隔20年左右，老的编程范型就会变得不足以应对一些组织和表达思想的新要求。新的范型必定会涌现出来，但这并不是一个简单的过程。每个新的编程范型都会引入一批编程语言，而不仅仅只是一种语言。最初的语言往往具有惊人的生命力，但也很不实用。比如面向对象编程语言Smalltalk或者函数式编程语言Lisp。接下来，其他范型的语言会加入一些新特性，允许开发人员在采用新概念的同时也可以安全地使用原先的老范型。例如Ada语言，它能够在过程式语言中使用一些面向对象的核心思想，比如封装。某些时候，一些混合语言恰恰是搭建在新老范型之间的一座实用的桥梁，比如C++。

① Edward Scissorhands，DVD 版，导演：Tim Burton（1990年）。发行商：加利福尼亚州比弗利山20世纪福克斯公司（2002年）。（译者注：这部电影中译名为《剪刀手爱德华》。Edward是一个机器人，有一双剪刀手。他被化妆品推销员带回家，开始在人类世界的生活。但由于他总是好心办坏事，邻居们逐渐无法接受这个单纯的Edward。最终Edward不得不躲到城堡里，用他的剪刀修剪他的植物和他的冰雕。）

紧接着，你将看到一些可用于商业应用的编程语言，比如Java或C#。最后，你才会看到新范型的一些成熟且完整的实现。

5.1.1　与Java的密切关系

Scala至少可以作为一座桥梁，也许还不仅如此。它与Java紧密集成，为人们提供了一个保护投资的机会，这体现在以下几个方面。

- Scala运行在Java虚拟机上，这使得Scala可以和现存的应用同时运行。
- Scala可以直接使用Java类库，使得开发人员可以利用现有的框架和遗留代码。
- Scala和Java一样都是静态类型语言，因此两种语言遵循一样的编程哲学。
- Scala的语法与Java比较接近，使得开发人员可以快速掌握语言基础。
- Scala既支持面向对象范型也支持函数式编程范型，这样开发人员就可以逐步在代码中运用函数式编程的思想。

5.1.2　没有盲目崇拜

一些编程语言过于迷信祖先，延展的概念有限，导致其基础不牢固。尽管与Java的相似性为人称道，然而Scala的设计仍然不乏有意义的新尝试，这样可以很好地满足其开发社区的需要。这些重要的改进代表了与Java语言的不同之处。

- *类型推断*。在Java中，你必须声明每个变量、实际参数或形式参数的类型。Scala则会在可能的情况下推断出变量的类型。
- *函数式编程概念*。Scala将函数式编程的重要概念引入Java。具体来说，这门语言可以通过不同方式使用已有的函数构造出新函数。在本章中你将看到的概念包括代码块、高阶函数（high-order function）以及一个复杂的集合库。Scala所提供的已远远超出基本语法糖的范畴了。
- *不变量*。Java的确允许使用不变量，不过是通过提供一个很少使用的修饰符实现的。在本章中，你将看到Scala会要求你明确地决定一个变量是否可变。这些决定将对应用程序在并发环境中的行为产生深远的影响。
- *高级程序构造*。Scala很好地使用了基础语言，并将有用的概念分层。在本章中我们将向你介绍用于并发应用的actor模型、使用高阶函数的Ruby风格的集合以及作为一等对象类型（first-class）的 XML的处理。

在开始学习Scala之前，我们应该了解一下Scala诞生背后的动机。我们将花些时间与Scala的缔造者在一起探讨一下他是如何决定将两种编程范型结合在一起的。

5.1.3　Martin Odersky访谈录

Martin Odersky，Scala的设计者，瑞士洛桑联邦理工学院（EPFL）的教授，该学院是瑞士两所联邦理工学院中的一所。他曾参与了Java泛型规范的制定，并且是javac编译器参考实现的作者。

他也是*Programming in Scala: A Comprehensive Step-by-Step Guide* [OSV08]一书的作者，这本书是目前市面上最好的Scala书籍之一。以下是对他的采访记录。

Bruce：你为什么要开发Scala?

Odersky博士：我坚信将函数式和面向对象两种编程范型统一起来将具有很大的实用价值。不过函数式编程社区对面向对象编程（OOP）不屑一顾的态度和面向对象程序员坚持函数式编程只是一种学术活动的信条都让我十分沮丧。因此，我想表明这两种模式可以统一，而且一些新的更强大的功能可能因此而产生。同时我也想设计出一门新语言，用它编写程序会让我自己感觉更加舒服。

Bruce：你最喜欢它哪一点呢?

Odersky博士：我喜欢它让程序员自由地表达自己并且感觉轻松自如，同时通过其类型系统，它还能为程序员提供强有力的支持。

Bruce：它最擅长解决什么样的问题?

Odersky博士：它实际上是一门通用语言。我会尝试用它去解决所有问题。即便如此，相对于其他主流语言，Scala具有一项独特优势，即对函数式编程的支持。因此所有函数式编程方法发挥重要作用的地方，Scala都会有出色的表现，无论是并发性和并发处理，还是处理XML的Web应用，或是实现领域特定语言。

Bruce：如果能让时光倒流，你想改变哪些特性?

Odersky博士：Scala的局部类型推断用起来很好，但是也有局限性。如果可以重新开始，我会尝试使用功能更强大的约束求解器（constraint solver）。也许现在依然可以做到这点，不过已有的大量用户群让这个工作变得更加困难。

5

Scala的拥趸越来越多，这是因为Twitter已经将其核心消息处理的实现从Ruby迁移到了Scala。面向对象的特性使得程序可以相当平滑地从Java语言迁移到Scala，但是Scala真正吸引大家眼球的概念是其函数式编程的特性。纯函数式语言实现了一种具有很强数学基础的编程风格。一门函数式语言具有以下几点特性。

- 函数式程序由函数组成。
- 函数总是具有返回值。
- 函数对于相同的输入总是会返回相同的值。
- 函数式程序禁止改变状态或修改数据。一旦你设置了一个值，就无需再管它了。

严格地讲，Scala并不是一门纯函数式编程语言，就像C++不是一门纯面向对象语言一样。它允许可变值，这可能会导致函数在输入相同的情况下输出却不同。[和大多数面向对象语言一样，使用`getter`（获取方法）和`setter`（设置方法）将破坏这一规则。]但是，它提供了让开发人员合理应用函数式抽象的方法。

5.1.4 函数式编程与并发

当前对研究并发编程的面向对象程序员来说，最大的问题在于可变的状态，也就是说数据是可以随时改变的。任何变量在初始化后都是可变的，可以被多次赋值。如果说可变状态是王牌大

贱谍的话，那并发就好比邪恶博士[①]。如果两个不同的线程可以在同一时间更改相同的数据，就很难保证执行过程中数据始终处于有效状态，测试也几乎是不可能的。数据库系统通过事务和锁来应对这个问题。面向对象编程语言则是通过给程序员提供控制对共享数据访问的工具来应对这个问题。但即便是知道如何使用这些工具，程序员通常也无法很好地使用它们。

函数式编程语言通过去除等式中的可变状态来解决这些问题。Scala不强迫你完全去除可变状态，不过它确实给你提供了一种使用纯函数风格编程的方法。

有了Scala，你就无需在Smalltalk和Lisp之间作出艰难的选择了。让我们开始用Scala代码将面向对象编程和函数式编程两个世界融合在一起吧。

5.2　第一天：山丘上的城堡

在《剪刀手爱德华》这部影片中，山丘上有一座看上去有些不同的城堡。在过去，这个城堡是一个神秘且迷人的地方，不过现在它却显得年代久远且荒废失修。冷风通过破碎的窗户刮进城堡，房间也早已不是曾经的那副模样了。曾经让人感到十分舒服的房子现在却显得冰冷甚至是令人生厌。面向对象编程范型同样也显示出了一些衰败的迹象，特别是早期实现的面向对象语言。Java语言的静态类型系统和并发机制的实现都已经过时，需要重新设计，改头换面。在这一节中，我们就在山丘上的房子中，即面向对象编程范型的背景下，谈谈Scala。

Scala运行在Java虚拟机（JVM，Java Virtual Machine）上。我不打算提供一份有关Java语言的详尽的概述，其他地方免费提供了这些资料。你将在Scala中看到一些Java的思想，不过我将努力减小这种影响，你无需同时学习两门语言。现在，安装Scala。我在本书中使用的是2.7.7最终版。

5.2.1　Scala 类型

当Scala安装完毕后，输入scala命令启动控制台。如果一切正常的话，你不会看到任何错误信息。你会看到一个scala>提示符，接下来就可以输入一些代码了。

```
scala> println("Hello, surreal world")
Hello, surreal world

scala> 1 + 1
res8: Int = 2

scala> (1).+(1)
res9: Int = 2

scala> 5 + 4 * 3
res10: Int = 17

scala> 5.+(4.*(3))
```

① 邪恶博士（Dr.Evil）是电影《王牌大贱谍》中的一个人物。《王牌大贱谍》是由Jay Roach导演的一部喜剧电影。在这部影片中，20世纪60年代的特工奥斯汀一次又一次地打败了邪恶博士，拯救了全世界。

```
res11: Double = 17.0

scala> (5).+((4).*(3))
res12: Int = 17
```

整数是对象。在Java中，在`int`（原生类型）和`Integer`（对象）之间做转换曾让我耗尽脑力。事实上，除了少数例外，Scala中一切都是对象。与大多数静态类型的面向对象语言相比，这是一个显著的不同。让我们看看Scala是如何处理字符串的：

```
scala> "abc".size
res13: Int = 3
```

字符串也是一等类型对象，并混合了一点语法糖。下面我们尝试强制产生一个类型冲突：

```
scala> "abc" + 4
res14: java.lang.String = abc4

scala> 4 + "abc"
res15: java.lang.String = 4abc

scala> 4 + "1.0"
res16: java.lang.String = 41.0
```

嗯，这可不太符合我们预想的结果。Scala将那些数字强制转换为字符串了。我们再加把劲儿强制产生一个错误匹配：

```
scala> 4 * "abc"
<console>:5: error: overloaded method value * with alternatives (Double)Double
<and> (Float)Float <and> (Long)Long <and> (Int)Int <and> (Char)Int
<and> (Short)Int <and> (Byte)Int cannot be applied to (java.lang.String)
       4 * "abc"
         ^
```

噢，这就对了。Scala实际上是强类型的。Scala使用类型推断，这样大多数情况下，它都能通过语法线索推断出变量的类型。但与Ruby不同的是，Scala可以在编译期间进行类型检查。Scala实际上是先对代码进行编译，然后再一行一行执行代码的。

另外，我知道你可能正想找回Java字符串类型。多数有关Scala的文章和书籍都会详细讨论这个话题，不过我们不会这样做，我们仍然继续研究程序构造，我想这对你来说才是最有吸引力的。现在，我要告诉你们，在很多地方，Scala会使用一种跨语言的类型管理策略。其中之一是在适合的地方使用简单的Java类型，比如`java.lang.String`。相信我，并接受这个过于简化的介绍吧。

5.2.2 表达式与条件

现在我们将通过例子严谨快速地学习一些基本语法。下面是一些Scala的`true/false`表达式：

```
scala> 5 < 6
res27: Boolean = true

scala> 5 <= 6
```

```
res28: Boolean = true

scala> 5 <= 2
res29: Boolean = false

scala> 5 >= 2
res30: Boolean = true

scala> 5 != 2
res31: Boolean = true
```

这里没有什么可讲的。通过之前几门语言的介绍你应该很熟悉这种C风格的语法了。接下来在`if`语句中使用一个表达式：

```
scala> val a = 1
a: Int = 1

scala> val b = 2
b: Int = 2

scala> if ( b < a) {
     |   println("true")
     | } else {
     |   println("false")
     | }
false
```

我们给两个变量赋了值，并在一个`if/else`语句中对它们进行了比较。我们来仔细看一下变量赋值操作。首先，注意这里并没有指定变量的类型。Scala与Ruby不同，它在编译期间绑定变量类型。不过Scala也与Java不同，它会推断出变量的类型，因此不需要输入`val a : Int = 1`，当然如果你想这么做的话，也是可以的。

接下来，注意这些Scala变量的声明以关键字`val`开始。当然你也可以使用关键字`var`，`val`用来声明不变量，而`var`不是。后面我们会详细讨论它们。

在Ruby中，0等价于`true`，而在C中，0则等价于`false`。在这两种语言中，`nil`都等价于`false`。让我们看看Scala是如何处理它们的吧。

```
scala> Nil
res3: Nil.type = List()

scala> if(0) {println("true")}
<console>:5: error: type mismatch;
 found   : Int(0)
 required: Boolean
       if(0) {println("true")}
          ^

scala> if(Nil) {println("true")}
<console>:5: error: type mismatch;
 found   : Nil.type (with underlying type object Nil)
```

```
required: Boolean
      if(Nil) {println("true")}
         ^
```

我们看到Nil是一个空列表，并且你甚至无法对Nil或0进行条件测试。这种行为与Scala的强类型和静态类型语言设计哲学相吻合。Nil和数字不是布尔值，所以不要按布尔值对待它们。有了简单表达式和最基本的判断结构之后，我们来学习循环。

5.2.3 循环

由于接下来的几个程序更加复杂，我们将选择使用脚本而不是控制台来运行它们。与Ruby和Io一样，你可以通过`scala path/to/program.scala`来运行脚本。

在第二天学习代码块时，你会看到多种在结果集中进行迭代的方法。不过目前，我们将主要学习命令式风格的循环。你会看到这与Java风格的循环结构很相似。

有关静态类型的内心斗争

一些初级的编程爱好者常常将强类型和静态类型这两个概念混淆。强类型是指这门语言检查两种类型是否兼容，如果不兼容则会抛出一个错误或强制类型转换，尽管上述说法不是很严格。表面上，Java 和 Ruby 都是强类型的。（我意识到这个想法有些过于简化了。）另一方面，汇编语言和 C 语言则是弱类型的。编译器并不关心在某一内存位置上的数据到底是一个整数、一个字符串还是只是一个普通数据。

静态类型和动态类型则是另外一个话题。静态类型语言强迫在类型结构的基础上执行多态。判断是否是一只鸭子的依据是其基因蓝图（静态），还是因其叫声和走路的姿态像一只鸭子（动态）。静态类型语言的好处在于编译器和工具等对你的代码更加了解，可以用于捕捉错误，突出显式代码以及便于重构代码。付出的代价则是你不得不做更多的工作并且会受到一些限制。作为开发者，你的开发经历经往往会决定你是如何权衡使用静态类型的。

我第一次是使用 Java 进行面向对象开发的。我看到一个又一个框架试图摆脱 Java 静态类型的束缚。这个行业在三个版本的企业级 Java 组件（EJB，Enterprise Java Beans）、Spring、Hibernate、JBOSS 以及面向方面编程（AOP，aspect-oriented programming）上投资了上千万美元，试图让某些应用模型具有更强的适应性。我们正在让 Java 的类型模型更趋动态化，并且这场斗争的每一步都是十分激烈的，感觉更像是对抗邪教，而不是为了改善编程环境。本书走的也是这条路，从日益增多的动态框架到动态语言。

我对静态类型的偏见被这场 Java 战争改变了。Haskell 及其强大的静态类型系统正在帮助我从阴影中缓慢地走出来。我问心无愧。你已经被邀请与一个不愿公开身份的政客一起吃顿便饭了，不过我会尽最大努力保证这次谈话是轻松且无偏见的。

5

首先是基本的while循环：

scala/while.scala

```
def whileLoop {
    var i = 1
    while(i <= 3) {
        println(i)
        i += 1
    }
}

whileLoop
```

这里定义一个函数。顺便说一句，Java开发者需要注意，这里无需明确指定public。在Scala中，public是默认的可见级别，即这个函数将对所有调用者可见。

在这个方法里，声明了一个循环次数为3的循环。i总被改变，所以我们用var声明它。然后你看到了一个Java风格的while语句声明。你可以看到括号里面的代码一直执行，直到条件表达式求值后的结果为false。你可以这样运行这段代码：

```
batate$ scala code/scala/while.scala
1
2
3
```

for循环的工作方式与Java和C中的也很相似，但语法稍有不同：

scala/for_loop.scala

```
def forLoop {
    println( "for loop using Java-style iteration" )
    for(i <- 0 until args.length) {
        println(args(i))
    }
}

forLoop
```

这里的参数是一个变量，后面跟着<-操作符，然后是用initialValue until endingValue的形式表示的循环范围。在这里，我们根据传入的命令行参数进行迭代：

```
batate$ scala code/scala/forLoop.scala its all in the grind
for loop using Java-style iteration
its
all
in
the
grind
```

与Ruby一样，你也可以使用循环对一个集合进行迭代。现在，使用foreach，它会让你联想到Ruby中的each：

scala/ruby_for_loop.scala

```
def rubyStyleForLoop {
    println( "for loop using Ruby-style iteration" )
    args.foreach { arg =>
        println(arg)
    }
}

rubyStyleForLoop
```

args是由传入的命令行参数构成的一个列表。Scala将每个列表元素一个接着一个的传入这个代码块。在我们的例子中，arg是args列表中的一个元素。在Ruby中，与此功能等价的代码是args.each{|arg| println(arg)}。指定每个参数的语法略有不同，但是思想是相同的。下面是实际运行的代码：

```
batate$ scala code/scala/ruby_for_loop.scala freeze those knees chickadees
for loop using Ruby-style iteration
freeze
those
knees
chickadees
```

稍后，你会发现自己会更多地使用这种迭代方法而不是其他命令式风格的循环。但是既然我们正将注意力集中在山丘上的那座房子上，那就将这方面的话题推迟一些。

5

5.2.4 范围与元组

与Ruby一样，Scala支持一等类型的范围（range）。启动控制台，输入下面这些代码：

```
scala> val range = 0 until 10
range: Range = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> range.start
res2: Int = 0

scala> range.end
res3: Int = 10
```

原来如此。它很像Ruby的范围。你也可以指定步长：

```
scala> range.step
res4: Int = 1

scala> (0 to 10) by 5
res6: Range = Range(0, 5, 10)

scala> (0 to 10) by 6
res7: Range = Range(0, 6)
```

Ruby的范围1..10相当于从1~10，而Ruby的范围1...10相当于1直到10。前者包含尾端点。

```
scala> (0 until 10 by 5)
res0: Range = Range(0, 5)
```

你也可以用下面方法指定范围变化的方向：

```
scala> val range = (10 until 0) by -1
range: Range = Range(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

但是方向是无法被推断出来的：

```
scala> val range = (10 until 0)
range: Range = Range()

scala> val range = (0 to 10)
range: Range.Inclusive = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

无论你为范围设置的结束点为多少，1都是默认步长。范围不限于整数：

```
scala> val range = 'a' to 'e'
range: RandomAccessSeq.Projection[Char] = RandomAccessSeq.Projection(a, b, c, d, e)
```

Scala会为你做一些隐式类型转换。事实上，当你指定了一个for语句，也就等于指定了一个范围。

和Prolog一样，Scala提供了元组。元组是一个固定长度的对象集合。你在许多其他函数式编程语言中也会发现这个模式。元组中的对象可以具有不同类型。在纯函数式编程语言中，程序员经常用元组表示对象以及它们的属性。看看这个例子：

```
scala> val person = ("Elvis", "Presley")
person: (java.lang.String, java.lang.String) = (Elvis,Presley)

scala> person._1
res9: java.lang.String = Elvis

scala> person._2
res10: java.lang.String = Presley

scala> person._3
<console>:6: error: value _3 is not a member of (java.lang.String, java.lang.String)
       person._3
              ^
```

Scala使用元组而不是列表进行多值赋值：

```
scala> val (x, y) = (1, 2)
x: Int = 1
y: Int = 2
```

由于元组具有固定长度，Scala可以基于每个元组元素的值对其进行静态类型检查：

```
scala> val (a, b) = (1, 2, 3)
<console>:15: error: constructor cannot be instantiated to expected type;
 found   : (T1, T2)
 required: (Int, Int, Int)
       val (a, b) = (1, 2, 3)
           ^
<console>:15: error: recursive value x$1 needs type
       val (a, b) = (1, 2, 3)
            ^
```

有了这些基础后，我们把它整合在一起，创造出一些面向对象的类定义。

5.2.5　Scala中的类

在Scala中，可以只用一行代码定义那些只有属性而没有方法或构造器的简单类：

```
class Person(firstName: String, lastName: String)
```

在定义一个简单的值类型时无需指定定义体。Person类是公有的（public），并具有firstName和lastName两个属性。

你可以在控制台中使用这个类：

```
scala> class Person(firstName: String, lastName: String)
defined class Person

scala> val gump = new Person("Forrest", "Gump")
gump: Person = Person@7c6d75b6
```

但这还远远不够。面向对象的类集数据和行为于一身。接下来，我们用Scala构建一个完整的面向对象的类。我们给这个类起名为Compass（罗盘）。罗盘初始指向北面。我们可以让这个罗盘向左或向右转动90°并且相应地更新指向。这里是关于Compass类的全部Scala代码：

scala/compass.scala

```
class Compass {

  val directions = List("north", "east", "south", "west")
  var bearing = 0
  print("Initial bearing: ")
  println(direction)

  def direction() = directions(bearing)

  def inform(turnDirection: String) {
    println("Turning " + turnDirection + ". Now bearing " + direction)
  }

  def turnRight() {
    bearing = (bearing + 1) % directions.size
    inform("right")
  }

  def turnLeft() {
    bearing = (bearing + (directions.size - 1)) % directions.size
    inform("left")
  }
}

val myCompass = new Compass

myCompass.turnRight
myCompass.turnRight
```

5

```
myCompass.turnLeft
myCompass.turnLeft
myCompass.turnLeft
```

这里的语法相对简单，并具备一些显著的特点。构造器负责定义实例变量（至少包括那些你没有传给构造器的变量）和方法。与Ruby不同，所有方法的定义都包含参数类型和名字，并且初始化代码块不包含在任何一个方法里面。让我们将代码分解，一步步详细介绍：

```
class Compass {

  val directions = List("north", "east", "south", "west")
  var bearing = 0

  print("Initial bearing: ")
  println(direction)
```

类定义后面的整个代码块实际上就是构造器。构造器包括一个方向（direction）列表和一个方位（bearing），方位仅仅是方向列表的下标。之后，转动操作会操纵方位。接下来，我们提供了一些方便的方法向类的使用者显示当前方位：

```
def direction() = directions(bearing)
def inform(turnDirection: String) {
  println("Turning " + turnDirection + ". Now bearing " + direction)
}
```

这个构造器继续进行方法定义。接下来是一个方法定义。`direction`方法只是将`directions`列表中对应`bearing`下标值的那个元素返回。Scala提供了一种单行方法定义的替代语法，使用这种语法可以省略那个括起方法定义体的括号。

当使用者转动罗盘，`inform`方法将输出一条友好的信息。它接受一个简单的参数，即转动的方向。这个方法没有返回值。下面让我们看看这些处理转动的方法吧。

```
def turnRight() {
  bearing = (bearing + 1) % directions.size
  inform("right")
}

def turnLeft() {
  bearing = (bearing + (directions.size - 1)) % directions.size
  inform("left")
}
```

这些处理转动的方法根据转动的方向改变方位值。%操作符是一个取模操作。（这个操作执行一个除法操作，忽略商值，只返回余数。）其结果是当罗盘向右转时，方位值加1；当罗盘向左转时，方位值减1。这些方法还对返回结果进行了适当地包装。

辅助构造器

你已经看到了基本构造器是如何工作的了。它是一个用于初始化类和方法的代码块。你也可以使用其他替代品。考虑一下`Person`类，它有两个构造器：

scala/constructor.scala

```
class Person(first_name: String) {
  println("Outer constructor")
  def this(first_name: String, last_name: String) {
    this(first_name)
    println("Inner constructor")
  }
  def talk() = println("Hi")

}

val bob = new Person("Bob")
val bobTate = new Person("Bob", "Tate")
```

这个类拥有一个只接受单一参数firstName的构造器和一个名为talk的方法。注意那个this方法，它是这个类的第二个构造器。它接受两个参数：firstName和lastName。一开始，这个方法通过this调用只有一个firstName参数的主构造器。

类定义后面的代码使用了两种方法实例化一个person，第一种方法使用了主构造器，第二种方法则是使用了辅助构造器。

```
batate$ scala code/scala/constructor.scala
Outer constructor
Outer constructor
Inner constructor
```

就是这么回事。辅助构造器非常重要，因为它们允许更为宽泛的使用模式。让我们看看如何创建类方法吧。

5.2.6　扩展类

到目前为止，这些类看起来平淡无奇。我们只是创建了一些仅仅包含了属性和方法的基本类。在本节中，我们将看到一些类之间交互的方法。

1. 伙伴对象和类方法

在Java和Ruby中，你会在类定义中同时定义类方法和实例方法。在Java中，类方法用static关键字修饰。Ruby则使用语法def self.class_method。Scala没有采用这两种策略。Scala会在类定义中声明实例方法。当一些类只能拥有一个实例时，可以使用object而不是class关键字定义这个类。下面是一个例子：

scala/ring.scala

```
object TrueRing {
 def rule = println("To rule them all")
}

TrueRing.rule
```

TrueRing的定义确实很像一个类的定义，但它实际上是创建了一个单件（singleton）对象。在Scala中，对象定义和类定义可以具有相同的名称。有了这个方案，你可以在一个单件对象的声明中创建类方法而在类声明中创建实例方法。在我们的例子中，rule方法是一个类方法。这个策略称为伙伴对象（companion objects）。

2. 继承

在Scala中继承相当简单易懂，但语法却不失精确。这里有一个用Employee类扩展Person类的例子。注意Employee的id字段中保存了一个额外的员工编号。下面是代码：

scala/employee.scala

```
class Person(val name: String) {
  def talk(message: String) = println(name + " says " + message)
  def id(): String = name
}


class Employee(override val name: String,
                        val number: Int) extends Person(name) {
  override def talk(message: String) {
    println(name + " with number " + number + " says " + message)
  }
  override def id():String = number.toString
}

val employee = new Employee("Yoda", 4)
employee.talk("Extend or extend not. There is no try.")
```

在这个例子中，我们用Employee类扩展了Person基类。我们在Employee类中增加了一个新的实例变量number，并且重写了talk方法以增加一些新的行为。多数复杂的语法都与类的构造器定义有关。注意，尽管你可以省略参数类型信息，但必须为Person类指定完整的参数列表。

无论是在构造器中还是在任何扩展基类的方法里，override关键字都是必需的。这个关键字可以防止你因无意中的拼写错误而引入新方法。总而言之，这里没有什么可以让你大吃一惊的。但有时，我会觉得这有点像Edward尝试抚摸一个脆弱的小兔子。我们继续……

3. trait

每种面向对象语言都必须解决这样的问题：一个对象可以拥有多种不同的角色。对象可以是一个可持久化、可序列化的“灌木丛”。你肯定不想让你的“灌木丛”知道如何将二进制数据存入MySQL数据库中。为了解决这个问题，C++使用了多重继承，Java使用了接口（interface），Ruby使用了mixins，而Scala使用了trait。Scala的trait与Ruby的mixin类似，用模块实现。或者如果你喜欢，也可以将Scala的trait看成是Java的接口外加一个接口的实现。我们将trait看成是一个部分类（partial-class）的实现。理想情况下，它应该可以帮你解决一个关键问题。下面是一个为Person类增加trait Nice的例子：

scala/nice.scala

```
class Person(val name:String)

trait Nice {
  def greet() = println("Howdily doodily.")
}

class Character(override val name:String) extends Person(name) with Nice

val flanders = new Character("Ned")
flanders.greet
```

你看到的第一个元素是Person类，它是只有一个名为name属性的简单类。第二个元素是名为Nice的trait，这就是mixin。它只有一个方法greet。最后一个元素是一个名为Character的类，并结合了trait Nice。使用者现在可以通过任何Character实例调用greet方法了。输出的结果和你预计的一样：

```
batate$ scala code/scala/nice.scala
Howdily doodily.
```

这里没有什么太复杂的内容。我们可以在任何Scala类中结合这个带有greet方法的Nice trait，并引入greet行为。

5.2.7 第一天我们学到了什么

由于我们需要使用同一种语言进行两种不同范型的开发，所以在第一天的学习中我们涵盖了Scala语言的大部分内容。第一天的学习告诉我们Scala支持面向对象概念，可运行在Java虚拟机上，并可以使用现有的Java库。Scala的语法与Java相似并且也是强类型和静态类型的。然而，Martin Odersky实现Scala是为了在面向对象编程和函数式编程两种范型之间搭建起一座桥梁。在第二天的学习中，我们会介绍函数式编程的概念，它会让并发应用程序设计起来更为简单。

Scala的静态类型也是可以推断出来的，用户无需在任何场合都显式声明变量的类型，因为Scala经常可以根据语法线索推断出这些类型。编译器也可以强制类型转换，比如整型转换为字符串，在合理的情况下，编译器还允许隐式类型转换。

Scala的表达式用法与其他语言非常相似，不过在Scala中更加严格一些。绝大多数条件表达式必须是布尔类型。0或Nil不能充当条件表达式，它们可用来代替非真非假。Scala的循环或控制结构与其他语言相比倒是没有什么显著的不同。Scala支持一些更高级的类型，比如元组（由不同类型组成的固定长度列表）和范围（固定不变的、包括所有端点元素的有序数字序列）。

Scala的类与Java中的类很相似，但是它们不支持类方法，而是使用了一种称为伙伴对象的概念将同一个类的类方法和实例方法混在一起。在Ruby使用mixin和Java使用接口的地方，Scala使用一个类似mixin的名为trait的结构。

在第二天的学习里，你会全面地学习Scala的函数式特性。你会学习代码块、集合、不变量以及一些高级的内置方法，如foldLeft。

5.2.8　第一天自习

第一天的Scala学习涵盖了很多内容，但是这些内容多是为大家所熟知的。这些面向对象的概念你也应该十分熟悉。这里的练习与早先书中的练习比起来略微有些深度，但是你应该是可以应付的。

找

- Scala的API。
- 对比Java与Scala。
- 关于`val`和`var`的讨论。

做

- 编写一个游戏，可以用X、O和空字符玩井字游戏（tic-tac-toe），检查是否有胜者，或是否不分胜负，或目前没有胜者。适当地使用类。
- 加分题：让两个选手玩井字游戏。

5.3　第二天：修剪灌木丛和其他新把戏

在《剪刀手爱德华》中，当Edward意识到他来自于山丘上的那座房子，并且他的独特能力可以让他在当下社会中获得一个特殊的位置时，他感觉一切就像做梦一样。

任何了解编程语言历史的人都曾经见识过这样的事情。当面向对象编程范型还是新鲜事物时，大家都无法接受Smalltalk，因为这个范型太新。我们需要的是一门既可以继续支持过程化编程，同时又可以进行面向对象思想编程试验的语言。C++新的面向对象方式可以与原有C语言的过程化特性很好地共存。结果是大家可以在老旧的上下文环境中开始使用新的编程手法。

现在是时候让Scala以函数式编程语言的身份接受考验了。有些用法初次看起来很别扭，不过其思想很重要，功能也很强大。它们将构成并发应用的基础，你将在第三天的学习中见识到这些并发应用。我们从一个简单的函数开始今天的学习：

```
scala> def double(x:Int):Int = x * 2
double: (Int)Int

scala> double(4)
res0: Int = 8
```

在Scala中定义一个函数与Ruby十分相似。`def`关键字既可以定义函数也可以定义方法。参数以及参数的类型紧随其后。然后，你可以指定一个可选的返回类型。Scala经常可以推断出返回结果的类型。

要调用该函数，只需使用函数名和参数列表即可。注意，这与Ruby不同，这里的括号在上下文当中是必需的。

这是单行方法定义。你也可以使用块形式来定义一个方法：

```
scala> def double(x:Int):Int = {
     |    x * 2
```

```
     | }
double: (Int)Int

scala> double(6)
res3: Int = 12
```

在返回类型Int后面的=是必需的。漏掉的话会出错。这是函数声明的主要形式。你也许会看到一些带有微小变化的函数声明形式，诸如省略参数，但这种形式是最常见的。

接下来继续来学习变量，你会在函数中使用到它们。如果你想了解纯函数式编程的模型，那么你就要专注于变量的生命周期。

5.3.1　对比var和val

Scala基于Java虚拟机，并且和Java有着紧密的关系。但在某些方面，这种设计目标限制了这门语言，而在其他方面，Scala可以充分利用过去15年或20年编程语言的发展。你会看到Scala支持并发编程的设计得到了越来越多的重视。但是如果你不遵守基本的设计原则，世上所有关于并发的特性都无法帮助你。可变的状态是糟糕的。当你声明变量时，应该使用不变量，这样可以避免状态冲突。在Java中，这意味着使用final关键字。而在Scala中，不变量意味着使用val而不是var。

```
scala> var mutable = "I am mutable"
mutable: java.lang.String = I am mutable

scala> mutable = "Touch me, change me..."
mutable: java.lang.String = Touch me, change me...

scala> val immutable = "I am not mutable"
immutable: java.lang.String = I am not mutable

scala> immutable = "Can't touch this"
<console>:5: error: reassignment to val
       immutable = "Can't touch this"
                 ^
```

var变量的值是可变的，而val变量的值则是不变的。在控制台中，为方便起见，你可以多次重复定义一个变量，即使你使用的是val。一旦你脱离控制台，重定义val变量会引发一个错误。

在某些方面，Scala引入var风格（var-style）变量以支持传统的命令式编程风格，但是如果你正在学习Scala，最好避免使用var，特别是当你希望设计出更好的并发程序时。这一基本设计理念是区别函数式编程与面向对象编程的关键要素：可变状态限制并发。

下面继续学习一些我最喜欢的函数式语言的特性：集合操作。

5.3.2　集合

函数式编程语言因其甚为好用的操作集合的特性而闻名已久。最早的函数式编程语言之一Lisp就是围绕处理列表的想法而设计的，这个名字就是LISt Processing的缩写。用函数式编程语言

5

可以很容易地构建出包含数据和代码的复杂结构。Scala的主要集合类型包括列表（list）、集（set）和映射（map）。

1. 列表

和大多数函数式编程语言一样，最实用的数据结构是列表。Scala中的列表类型为List，它是一类事物的有序集合，可随机访问。在控制台中输入这些列表：

```
scala> List(1, 2, 3)
res4: List[Int] = List(1, 2, 3)
```

注意，第一行的返回值：List[Int] = List(1, 2, 3)。这个值不仅表明了整个列表的类型，而且表明了列表中的数据结构类型。一个字符串列表如下所示：

```
scala> List("one", "two", "three")
res5: List[java.lang.String] = List(one, two, three)
```

如果你在这里看到了一些Java的影子，那你是对的。Java有一个特性叫做泛型（generics），这种特性可以在列表或数组这样的数据结构中输入任意元素。当你有一个结合了字符串和整型数的列表时，让我们来看一下会发生什么：

```
scala> List("one", "two", 3)
res6: List[Any] = List(one, two, 3)
```

这里得到了数据类型Any，它是Scala中的一个通用数据类型。下面是一个访问列表中元素的例子：

```
scala> List("one", "two", 3)(2)
res7: Any = 3

scala> List("one", "two", 3)(4)
java.util.NoSuchElementException: head of empty list
        at scala.Nil$.head(List.scala:1365)
        at scala.Nil$.head(List.scala:1362)
        at scala.List.apply(List.scala:800)
        at .<init>(<console>:5)
        at .<clinit>(<console>)
        at RequestResult$.<init>(<console>:3)
        at RequestResult$.<clinit>(<console>)
        at RequestResult$result(<console>)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Met...
```

代码中使用了()操作符。列表访问是一个函数，所以使用的是()而不是[]。与Java和Ruby一样，Scala的列表下标也从0开始。不过与Ruby不同的是，访问超出列表下标范围的元素时会抛出异常。

你可以用负数作为下标，早期的Scala版本会返回列表的第一个元素：

```
scala> List("one", "two", 3)(-1)
res9: Any = one

scala> List("one", "two", 3)(-2)
res10: Any = one
```

```
scala> List("one", "two", 3)(-3)
res11: Any = one
```

由于这种行为与表示下标值过大的异常NoSuchElement有些许不一致，所以2.8.0版本修改了这种行为，让它返回java.lang.IndexOutOfBoundsException异常了。

最后提示一下，Scala中的Nil是一个空列表：

```
scala> Nil
res33: Nil.type = List()
```

当介绍代码块的时候，我们会使用列表作为基本的组成部分。不过现在，请先忍耐一下。我打算先介绍一些其他的集合类型。

2. 集

集与列表类似，不过集没有任何显式的顺序。你可以通过Set关键字指定一个集。

```
scala> val animals = Set("lions", "tigers", "bears")
animals: scala.collection.immutable.Set[java.lang.String] =
    Set(lions, tigers, bears)
```

从集中增删元素很容易：

```
scala> animals + "armadillos"
res25: scala.collection.immutable.Set[java.lang.String] =
    Set(lions, tigers, bears, armadillos)

scala> animals - "tigers"
res26: scala.collection.immutable.Set[java.lang.String] = Set(lions, bears)

scala> animals + Set("armadillos", "raccoons")
<console>:6: error: type mismatch;
 found   : scala.collection.immutable.Set[java.lang.String]
 required: java.lang.String
       animals + Set("armadillos", "raccoons")
                 ^
```

记住，集操作是没有破坏性的。每个集操作都会建立一个新的集而不是修改旧的集。默认情况下，集是不可改变的。你可以看到从集里增删一个元素轻而易举，但是你无法像Ruby那样通过+或-运算符合并集。在Scala中，使用++和--运算符来完成集的并和差操作。

```
scala> animals ++ Set("armadillos", "raccoons")
res28: scala.collection.immutable.Set[java.lang.String] =
  Set(bears, tigers, armadillos, raccoons, lions)

scala> animals -- Set("lions", "bears")
res29: scala.collection.immutable.Set[java.lang.String] = Set(tigers)
```

你也可以使用**[①]来完成集的交操作（返回两个集中相同的元素组成的新集）：

```
scala> animals ** Set("armadillos", "raccoons", "lions", "tigers")
res1: scala.collection.immutable.Set[java.lang.String] = Set(lions, tigers)
```

① 从Scala 2.8.0版本开始使用&，**已经被废弃。

5

与列表不同，集与次序无关。这意味着集相等与列表相等是两个不同的概念：

```
scala> Set(1, 2, 3) == Set(3, 2, 1)
res36: Boolean = true

scala> List(1, 2, 3) == List(3, 2, 1)
res37: Boolean = false
```

到目前为止，集操作已经介绍的足够多了。接下来介绍映射。

3. 映射

映射是一个键值（key-value）对，类似于Ruby的散列（Hash）。它的语法你也应该很熟悉：

```
scala> val ordinals = Map(0 -> "zero", 1 -> "one", 2 -> "two")
ordinals: scala.collection.immutable.Map[Int,java.lang.String] =
  Map(0 -> zero, 1 -> one, 2 -> two)

scala> ordinals(2)
res41: java.lang.String = two
```

与Scala的列表和集一样，你可以通过Map关键字指定一个映射。使用->操作符将映射的两个元素隔开。上面的代码中使用了一些语法糖，使得创建一个Scala映射变得很容易。接下来使用另外一种形式的散列映射，并指定key和value的类型：

```
scala> import scala.collection.mutable.HashMap
import scala.collection.mutable.HashMap
scala> val map = new HashMap[Int, String]
map: scala.collection.mutable.HashMap[Int,String] = Map()

scala> map += 4 -> "four"

scala> map += 8 -> "eight"

scala> map
res2: scala.collection.mutable.HashMap[Int,String] =
  Map(4 -> four, 8 -> eight)
```

首先，我们为可变HashMap导入一个Scala库。这意味着这个散列映射中的值是可以改变的。接下来，声明一个名为map的不变量。这意味着这个map的引用无法改变。注意，我们还指定了映射的类型。最后，向这个散列映射中添加一些键值对，并返回结果。

如果你指定了错误的类型，将显示以下错误信息：

```
scala> map += "zero" -> 0
<console>:7: error: overloaded method value += with alternatives (Int)map.MapTo
    <and> ((Int, String))Unit cannot be applied to ((java.lang.String, Int))
        map += "zero" -> 0
            ^
```

和预想的一样，Scala报告了一个类型错误。无论是在编译阶段还是在运行阶段，类型约束都是强制执行的。现在你已经了解了有关集合的基本语法和操作，下面我们深入了解一些细节。

4. Any和Nothing

在介绍匿名函数之前，先来了解一下Scala中的类层次关系。当你结合Java使用Scala时，你会

经常关注Java的类层次关系。不过，你也应该知道一些有关Scala类型的内容。Any是Scala类层次体系中的根类。这常常难以理解，不过你要知道所有的Scala类型都继承自Any类。

同样，Nothing类是所有类型的子类。譬如，对一个返回集合的函数来说，函数也可以返回Nothing，这与给定函数的返回值类型相符。Scala的类层次体系如图5-1所示。所有类都继承自Any，并且Nothing类继承自所有类。

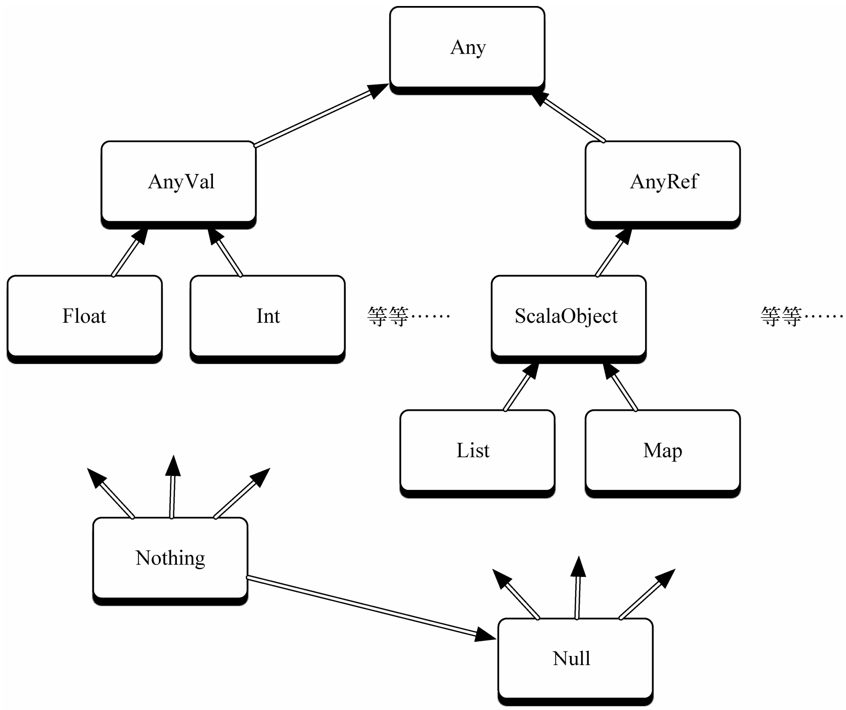


图5-1　Any和Nothing

当你处理nil的概念时，这里有些细微的差别。Null是一个trait，null则是Null的一个实例，与Java中的null类似，意思是一个空值。一个空集合是Nil，而Nothing是一个trait，是所有类的子类。Nothing类没有实例，所以不能像Null那样对其解引用（dereference）。例如，抛出异常的方法的返回值类型为Nothing，意思是根本没有返回值。

记住这些规则，你就会一路顺利。接下来我们将在集合上使用一些高阶函数。

5.3.3　集合与函数

在开始研究那些函数式基础更加坚实的语言之前，我想先正式介绍一些我们一直在使用的概念。首当其冲的概念就是高阶（high-order）函数。

与Ruby和Io一样，有了高阶函数，Scala的集合就会变得更加有吸引力。正如Ruby使用each以及Io使用foreach一样，Scala可以将函数传递给foreach。这个你一直使用的基础概念就是高

阶函数。通俗地说，高阶函数就是一个生成或使用函数的函数。更具体点说，高阶函数是一个以其他函数作为输入参数或以函数作为返回结果的函数。这种使用其他函数来构造函数的方法是函数式编程语言家族中的关键概念，并且它还会影响使用其他语言编写代码的方式。

Scala对高阶函数提供了强大的支持。我们没有时间去看一些高级主题了，诸如偏应用函数（partially applied function）或柯里化（currying），但是我们会学习将简单函数（也常常被称为代码块）作为参数传递给集合的方法。你可以把一个函数赋给任意一个变量或参数，也可以将它们作为参数传给函数，还可以将它们作为函数返回值返回。我们会集中介绍一下匿名函数，它们将作为输入参数传递给一些更为有趣的针对集合的方法。

1. foreach

我们要研究的第一个函数是foreach，它是Scala中的主要迭代方法。与Io一样，针对集合，foreach方法接受一个代码块作为参数。在Scala中，你可以用variableName => yourCode这样形式来表示代码块：

```
scala> val list = List("frodo", "samwise", "pippin")
list: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.foreach(hobbit => println(hobbit))
frodo
samwise
pippin
```

hobbit => println(hobbit)是一个匿名函数，即没有名字的函数。在这个声明中，=>的左边是参数，右侧是代码。foreach调用这个匿名函数，并将列表中的每个元组作为输入参数传递给匿名函数。你可能已经猜到了，你也可以针对集合映射使用相同的技术，尽管元素的次序无法保证。

```
scala> val hobbits = Set("frodo", "samwise", "pippin")
hobbits: scala.collection.immutable.Set[java.lang.String] =
  Set(frodo, samwise, pippin)

scala> hobbits.foreach(hobbit => println(hobbit))
frodo
samwise
pippin

scala> val hobbits = Map("frodo" -> "hobbit",
  "samwise" -> "hobbit", "pippin" -> "hobbit")
hobbits: scala.collection.immutable.Map[java.lang.String,java.lang.String] =
  Map(frodo -> hobbit, samwise -> hobbit, pippin -> hobbit)

scala> hobbits.foreach(hobbit => println(hobbit))
(frodo,hobbit)
(samwise,hobbit)
(pippin,hobbit)
```

当然了，映射会返回元组而不是元组中的元素。你应该还记得，foreach可以访问元组的任意一端，像这样：

```
scala> hobbits.foreach(hobbit => println(hobbit._1))
frodo
samwise
pippin

scala> hobbits.foreach(hobbit => println(hobbit._2))
hobbit
hobbit
hobbit
```

有了这些匿名函数，你可以做的将远远不止这些迭代。我先带你了解一些基础知识，然后学习Scala将函数与集合结合使用的其他一些有趣的方法。

2. 更多列表方法

我要先在这里简短地停留一下，先介绍几个有关操作列表的方法。这些基本的方法提供了在进行手工迭代或递归列表时所需的特性。首先，下面这些方法用于测试列表是否为空或检查列表的大小：

```
scala> list
res23: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.isEmpty
res24: Boolean = false

scala> Nil.isEmpty
res25: Boolean = true

scala> list.length
res27: Int = 3

scala> list.size
res28: Int = 3
```

注意，你既可以使用`length`也可以使用`size`去检查一个列表的大小。另外还要记住，`Nil`是一个空列表。与Prolog一样，获取列表的头和尾对递归非常有用。

```
scala> list.head
res34: java.lang.String = frodo

scala> list.tail
res35: List[java.lang.String] = List(samwise, pippin)

scala> list.last
res36: java.lang.String = pippin

scala> list.init
res37: List[java.lang.String] = List(frodo, samwise)
```

这真让人吃惊。你可以使用`head`和`init`从头开始递归列表，或使用`last`和`tail`从尾部开始递归列表。我们后续会更多地使用递归。下面用几个有趣且简便的方法来学习它们的使用方法：

5

```
scala> list.reverse
res29: List[java.lang.String] = List(pippin, samwise, frodo)

scala> list.drop(1)
res30: List[java.lang.String] = List(samwise, pippin)

scala> list
res31: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.drop(2)
res32: List[java.lang.String] = List(pippin)
```

这些方法的行为和预期的一样。reverse返回一个倒序列表，drop(*n*)返回一个前*n*个元素被删除后的列表，但不修改原列表。

3. 计数、映射、过滤以及其他

与Ruby一样，Scala拥有许多采用不同方式操作列表的方法。你可以用给定的条件过滤一个列表，用任何你想要的标准排序列表，用列表中的各个元素作为输入来创建其他列表，并且你还可以创建聚合值。

```
scala> val words = List("peg", "al", "bud", "kelly")
words: List[java.lang.String] = List(peg, al, bud, kelly)

scala> words.count(word => word.size > 2)
res43: Int = 3

scala> words.filter(word => word.size > 2)
res44: List[java.lang.String] = List(peg, bud, kelly)

scala> words.map(word => word.size)
res45: List[Int] = List(3, 2, 3, 5)
scala> words.forall(word => word.size > 1)
res46: Boolean = true

scala> words.exists(word => word.size > 4)
res47: Boolean = true

scala> words.exists(word => word.size > 5)
res48: Boolean = false
```

一开始，声明一个Scala列表，然后计算有多少个长度大于2的单词。count方法调用代码块word => word.size > 2，对每个列表中的元素按word.size > 2进行评估。count方法可以计算出所有求值结果为真的表达式的个数。

用同样方法，words.filter(word=>word.size>2)返回列表中所有长度大于2的元素所组成的新列表，这很像Ruby中的select方法。用同样的模式，map使用列表中所有单词的长度值组成了一个新的列表。如果代码块对于集合中的所有元素都返回true的话，那么forall返回true。如果代码块仅对集合中的某一个元素返回true，那么exists方法返回true。

有时，你可以使用代码块来泛化一个特性以得到更为强大的功能。例如，你可能需要使用传统方法对集合进行排序：

```
scala> words.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)
res49: List[java.lang.String] = List(al, bud, kelly, peg)
```

这里的代码使用了一个代码块，代码块有两个参数s和t。通过sort[①]你可以按照任何你想要的方式对两个参数进行比较。在前面的代码中，将字符转换为小写形式[②]然后比较它们。这将产生一个不区分大小写的搜索。我们也可以使用同样的方法通过比较单词的长度对列表进行排序。

```
scala> words.sort((s, t) => s.size < t.size)
res50: List[java.lang.String] = List(al, bud, peg, kelly)
```

通过使用代码块，可以基于任何想要的策略对集合进行排序。下面来看一个更为复杂的例子——foldLeft。

4. foldLeft

Scala中的foldLeft方法与Ruby中的inject方法非常相似。你只需提供一个初始值以及一个代码块，foldLeft就会将数组中的每个元素和另外的一个值传递给代码块。这里提到的另外的值可以是初始值（在第一次调用代码块时）也可以是从代码块中返回的结果（在后续的代码块调用时）。这个方法有两个版本。第一个版本/:是一个操作符，采用initialValue /:List codeBlock的形式，下面是一个使用这个方法的例子：

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> val sum = (0 /: list) {(sum, i) => sum + i}
sum: Int = 6
```

在介绍Ruby时我们曾详细地解释过这个语句序列，不过再看一遍也许仍会对你有所帮助。下面将说明它是如何工作的。

- 调用这个操作符，传入一个初始值和一个代码块。这个代码块有两个参数sum和i。
- 开始，/:将初始值0和列表的第一个元素1作为参数传给代码块。sum等于0，i等于1，这样0+1的结果等于1。
- 接下来，/:将从代码块返回的结果1返送给算式中的sum，这样sum等于1，i等于列表的下一个元素值2，这样代码块的执行结果为3。
- 最后，/:将从代码块返回的结果3返送给算式中的sum，这样sum等于3，i等于列表中的下一个元素值3，sum + i等于6。

另外一个版本的foldLeft的语法看起来很奇怪。它使用了一个被称为柯里化（currying）的概念。函数式编程语言使用柯里化将一个带有多个参数的函数转换为多个拥有独自参数列表的函数。我们将在第8章中介绍更多关于柯里化的内容。这里只要明白幕后实际是一个函数的组合而

① 在2.8.0版本中，sort已经被废弃，要使用sortWith代替它。——原书注

② 在2.8.0版本中，toLowerCase已经被废弃，要使用toLower代替它。——原书注

5

不仅仅是一个单独的函数即可。尽管这两个版本在机制和语法上有不同，但是得到的结果却是一致的：

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> list.foldLeft(0)((sum, value) => sum + value)
res54: Int = 6
```

注意，函数调用`list.foldLeft(0)((sum, value) => sum + value)`有两个参数列表。这就是我之前提到的柯里化概念。在本书后续的其他所有语言中你会看到这个方法的诸多版本。

5.3.4　第二天我们都学到了什么

第一天介绍了诸多你已经熟悉的面向对象特性。第二天介绍了Scala存在的主要原因：函数式编程。

我们由一个简单的函数开始。Scala有着灵活的函数定义的语法。编译器常常能推断出函数的返回类型，函数定义有单行和代码块两种形式，并且参数列表可以改变。

接下来，介绍了各种不同的集合。Scala支持三种集合：列表、映射和集。集是一个对象的集合。列表是一个有序集合。映射是一些键值对的集合。和Ruby一样，你看到了将代码块和各种不同的集合结合在一起所表现出的强大功能。我们看到了一些象征函数编程范型的集合API。

对于列表，也可以使用Lisp风格那样，使用`head`方法来返回列表的第一个元素`tail`方法返回剩余元素的列表，就像Prolog一样。我们也使用了`count`、`empty`和`first`方法来满足各种需要。不过功能最强大的方法是可以接受函数块作为参数。

我们使用`foreach`进行迭代并且使用过滤器有选择性地从列表中返回各种元素。我们还学会了使用`foldLeft`累加返回值，就像在一个集合中通过迭代来做一些事情一样，比如连续进行累加求和。

函数式编程主要是学习使用更高层次的结构而不是用Java风格的迭代去操作集合。我们将在第三天里充分展现这些技术，到时候将学习如何使用并发、处理XML以及做一些简单的贴近实际的练习，敬请关注。

5.3.5　第二天自习

我们已经深入研究了Scala，你即将开始看到它函数式编程的一面。每当你应对函数时，集合都是一个不错的起点。下面这些练习会使用到一些集合，当然也包括一些函数。

找

- 关于如何使用Scala文件的讨论。
- 闭包（closure）与代码块有何不同。

做

- 使用`foldLeft`方法计算一个列表中所有字符串的总长度。
- 编写一个`Censor trait`，包含一个可将`Pucky`和`Beans`替换为`Shoot`和`Darn`的方法。使用

映射存储脏话和它们的替代品。

- 从一个文件中加载脏话或它们的替代品。

5.4 第三天：剪断绒毛

就在《剪刀手爱德华》这部影片的高潮到来之前，Edward学会了在日常生活中像一位艺术家一样灵活地运用他那双剪刀手。他将灌木修剪成恐龙形状，用维达·沙宣[1]般的技艺毫不费力地精心制作出了令人惊叹的发型，甚至将家里的烤肉做成雕刻品。Scala让我们经历过了一些尴尬的时刻，不过当这种语言被放在合适的场合中时，它就会让你惊叹。像XML和并发这样的难题，几乎都会变得像例行公事般简单。让我们一起看一下吧。

5.4.1 XML

在解决现代编程问题的过程中我们越来越多地用到了XML（Extensible Markup Language，可扩展标记语言）。Scala将XML抬高到语言的一等编程结构，你可以像表示字符串那样来表示XML。

```
scala> val movies =
     | <movies>
     |     <movie genre="action">Pirates of the Caribbean</movie>
     |     <movie genre="fairytale">Edward Scissorhands</movie>
     | </movies>
movies: scala.xml.Elem =
<movies>
           <movie genre="action">Pirates of the Caribbean</movie>
           <movie genre="fairytale">Edward Scissorhands</movie>
       </movies>
```

在你使用XML定义完movies变量后，你就可以直接访问其中的不同元素了。

例如，要想查看其内部的所有文本，你只需输入：

```
scala> movies.text
res1: String =

           Pirates of the Caribbean
           Edward Scissorhands
```

从前一个例子中你看到了该变量的所有内部文本，但是并未受到只能一次使用全部文本的限制。我们可以更有选择性地使用。Scala内置了一种与XPath类似的查询语言，XPath是一种XML查询语言。不过由于在Scala中，关键字//是注释的修饰符，Scala将使用\和\\。为了搜索到顶层的节点，你可以使用单反斜线，如下所示：

```
scala> val movieNodes = movies \ "movie"
movieNodes: scala.xml.NodeSeq =

<movie genre="action">Pirates of the Caribbean</movie>
<movie genre="fairytale">Edward Scissorhands</movie>
```

---

① 维达·沙宣（Vidal Sassoon）是国际美发大师。——编者注

在上面的搜索中，我们查找到了XML的movie元素。你可以通过下标得到单个节点的信息：

```
scala> movieNodes(0)
res3: scala.xml.Node = <movie genre="action">Pirates of the Caribbean</movie>
```

我们刚刚找到零号元素，即Pirates of the Caribbean。你也可以使用@符号查找单个XML节点的属性。例如，执行以下搜索来找到文档中第一个元素的genre属性。

```
scala> movieNodes(0) \ "@genre"
res4: scala.xml.NodeSeq = action
```

这个例子仅仅是浅尝辄止，但是你知道该怎样去做了。如果将Prolog风格的模式匹配加进来，那么事情就会变得更加令人兴奋。下面就来看一个简单字符串模式匹配的例子。

5.4.2　模式匹配

模式匹配（pattern matching）允许你基于一些数据片断有条件地执行代码。Scala经常使用模式匹配，诸如当你解析XML或在线程间传递消息时。

下面是一个最简单的模式匹配的形式：

scala/chores.scala

```
def doChore(chore: String): String = chore match {
    case "clean dishes" => "scrub, dry"
    case "cook dinner" => "chop, sizzle"
    case _ => "whine, complain"
  }
  println(doChore("clean dishes"))
  println(doChore("mow lawn"))
```

我们定义了两个chore（日常事务）：clean dishes（刷盘子）和cook dinner（做饭）。紧邻每个chore都有一个代码块。在这个例子中，代码块只是简单地返回一个字符串。我们定义的最后一个chore是“_”，它是一个通配符。Scala执行第一个匹配成功的chore所对应的代码块，如果没有chore可以匹配成功，则返回"whine, complain"，如下所示：

```
>> scala chores.scala
scrub, dry
whine, complain
```

1. 哨兵

模式匹配也有一些装饰品。在Prolog中，模式匹配往往具有关联的条件。为了用Scala实现一个阶乘，我们在哨兵（Guard）中为每个匹配语句指定了一个条件：

scala/factorial.scala

```
def factorial(n: Int): Int = n match {
    case 0 => 1
    case x if x > 0 => factorial(n - 1) * n
```

```
}

println(factorial(3))
println(factorial(0))
```

第一个模式匹配是一个0，不过第二个哨兵的形式为`case x if x > 0`。它可以匹配任意大于0的x。通过这种方式你可以指定各种各样的条件。模式匹配也可以匹配正则表达式和类型。在随后的并行程序示例中，我们会定义一些空的类并用这些类作为消息使用。

2. 正则表达式

Scala中的正则表达式（Regular Expression）是一等类型。针对一个字符串的`.r`方法可以将任意字符串转换成正则表达式。在下一页里有一个正则表达式的示例，这个正则式可以匹配任何以大写或小写F开头的字符串。

```
scala> val reg = """^(F|f)\w*""".r
reg: scala.util.matching.Regex = ^(F|f)\w*

scala> println(reg.findFirstIn("Fantastic"))
Some(Fantastic)

scala> println(reg.findFirstIn("not Fantastic"))
None
```

我们从一个简单字符串开始。用"""为字符串划定界限，允许多行字符串，去除了对其内部字符串的求值过程。`.r`方法将字符串转换为正则表达式。然后使用`findFirstIn`方法找到第一次匹配成功的地方。

```
scala> val reg = "the".r
reg: scala.util.matching.Regex = the

scala> reg.findAllIn("the way the scissors trim the hair and the shrubs")
res9: scala.util.matching.Regex.MatchIterator = non-empty iterator
```

在这个例子中，构建了一个正则表达式，并使用`findAllIn`方法在字符串`"the way the scissors trim the hair and the shrubs"`中找到所有与`the`匹配的地方。如果需要，可以使用`foreach`遍历这个匹配结果列表，仅此而已。你可以像使用字符串那样去使用正则表达式来进行匹配。

3. XML和匹配

在Scala中将XML语法和模式匹配结合在一起是极具吸引力的。你可以浏览XML文件，并根据各种不同的XML元素有条件地执行代码。例如，考虑下面的XML文件movies：

scala/movies.scala

```
val movies = <movies>
    <movie>The Incredibles</movie>
    <movie>WALL E</movie>
    <short>Jack Jack Attack</short>
    <short>Geri's Game</short>
```

```
</movies>

(movies \ "_").foreach { movie =>
    movie match {
        case <movie>{movieName}</movie> => println(movieName)
        case <short>{shortName}</short> => println(shortName + " (short)")
    }
}
```

它查询了树中的所有节点。然后，它使用模式匹配去匹配short和movie节点。我喜欢这种方式，Scala通过使用XML语法、模式匹配和类XQuery语言使得最常见的任务变得极为简单，几乎不费吹灰之力。这里只是初步介绍了一些模式匹配的内容，在下一节关于并发的学习中，你会看到一些实际应用模式匹配的例子。

5.4.3　并发

Scala最重要的方面之一就是其处理并发的方式。其主要结构包括actor和消息传递。actor拥有线程池和队列池。当发送一条消息给actor时（使用!操作符），是将一个对象放到该actor的队列中。actor读取消息并采取行动。通常情况下，actor通过模式匹配器去检测消息并执行相应的消息处理。我们看看下面这个kids程序：

scala/kids.scala

```
import scala.actors._
import scala.actors.Actor._

case object Poke
case object Feed

class Kid() extends Actor {
  def act() {
    loop {
      react {
        case Poke => {
          println("Ow...")
          println("Quit it...")
        }
        case Feed => {
          println("Gurgle...")
          println("Burp...")
        }
      }
    }
  }
}

val bart = new Kid().start
val lisa = new Kid().start
```

```
println("Ready to poke and feed...")
bart ! Poke
lisa ! Poke
bart ! Feed
lisa ! Feed
```

在这个程序中，创建了两个空的、不重要的单件对象Poke和Feed。它们什么都不做，只是简单地作为消息使用。程序的主要部分是Kid类。Kid是actor，这意味着它将在线程池中的某个线程中运行，并从一个队列中读取消息。它将一个接着一个地处理每条消息。我们使用了一个简单的循环，循环里面是一个react结构，react可以接收来自actor的消息。模式匹配可以让我们匹配到适当的消息，或是Poke或是Feed。

脚本的后续部分创建了两个kid并通过向它们发送Poke或Feed消息来操作它们。你可以像下面这样来运行它：

```
batate$ scala code/scala/kids.scala
Ready to poke and feed...
Ow...
Quit it...
Ow...
Quit it...
Gurgle...
Burp...
Gurgle...
Burp...

batate$ scala code/scala/kids.scala
Ready to poke and feed...
Ow...
Quit it...
Gurgle...
Burp...
Ow...
Quit it...
Gurgle...
Burp...
```

我多次运行了这个程序，以表明它确实是并发的。注意多次运行结果输出的次序是不同的。使用actor，你还可以设置超时处理（reactWithin），当在指定时间内没有接收到消息时，会触发超时处理。此外，你还可以使用receive（它将阻塞线程）和receiveWithin（它将在设置的超时时间内阻塞线程）。

5.4.4 实际中的并发

例子中的模拟程序影响力毕竟是有限的，让我们做一些更有力的事情吧。在这个名为sizer的程序中，我们将计算网页的大小。我们点击一些页面，然后计算它们的大小。由于需要很长的等待时间，所以我们想使用actor以并发的方式抓取所有网页。先看一下完整的程序吧。然后再来看看个别的代码段：

scala/sizer.scala

```
import scala.io._
import scala.actors._
import Actor._

object PageLoader {
 def getPageSize(url : String) = Source.fromURL(url).mkString.length
}

val urls = List("http://www.amazon.com/",
               "http://www.twitter.com/",
               "http://www.google.com/",
               "http://www.cnn.com/" )

def timeMethod(method: () => Unit) = {
 val start = System.nanoTime
 method()
 val end = System.nanoTime
 println("Method took " + (end - start)/1000000000.0 + " seconds.")
}

def getPageSizeSequentially() = {
 for(url <- urls) {
   println("Size for " + url + ": " + PageLoader.getPageSize(url))
 }
}

def getPageSizeConcurrently() = {
 val caller = self

 for(url <- urls) {
   actor { caller ! (url, PageLoader.getPageSize(url)) }
 }

 for(i <- 1 to urls.size) {
   receive {
     case (url, size) =>
       println("Size for " + url + ": " + size)
   }
 }
}

println("Sequential run:")
timeMethod { getPageSizeSequentially }

println("Concurrent run")
timeMethod { getPageSizeConcurrently }
```

那么先从顶部开始。先为actors和io导入一些基本库，以便可以处理并发并且构造HTTP请

求。接下来，计算一个给定URL的Web页面大小：

```
object PageLoader {
 def getPageSize(url : String) = Source.fromURL(url).mkString.length
}
```

然后，定义了一个变量，其值为一些URL。之后，定义了一个方法来测定每个Web页面请求所消耗的时间：

```
def timeMethod(method: () => Unit) = {
 val start = System.nanoTime
 method()
 val end = System.nanoTime
 println("Method took " + (end - start)/1000000000.0 + " seconds.")
}
```

接下来，用两种不同的方法发起Web页面请求。第一种是按顺序的，在一个for循环中以迭代的方式发起每个请求。

```
def getPageSizeSequentially() = {
 for(url <- urls) {
   println("Size for " + url + ": " + PageLoader.getPageSize(url))
 }
}
```

下面是异步发起Web请求的方法：

```
def getPageSizeConcurrently() = {
 val caller = self

 for(url <- urls) {
   actor { caller ! (url, PageLoader.getPageSize(url)) }
 }

 for(i <- 1 to urls.size) {
   receive {
     case (url, size) =>
       println("Size for " + url + ": " + size)
   }
 }
}
```

在这个actor中，我们将收到一组固定的消息。在foreach循环中，发送了4个异步请求。这些请求几乎是同时发出。接下来，简单地使用receive接收这4条消息。这就是实际工作中会使用到的方法。最后，准备好运行这个脚本并启动这个测试：

```
println("Sequential run:")
timeMethod { getPageSizeSequentially }
println("Concurrent run")
timeMethod { getPageSizeConcurrently }
```

下面是输出结果：

5

```
>> scala sizer.scala
Sequential run:
Size for http://www.amazon.com/: 81002
Size for http://www.twitter.com/: 43640
Size for http://www.google.com/: 8076
Size for http://www.cnn.com/: 100739
Method took 6.707612 seconds.
Concurrent run
Size for http://www.google.com/: 8076
Size for http://www.cnn.com/: 100739
Size for http://www.amazon.com/: 84600
Size for http://www.twitter.com/: 44158
Method took 3.969936 seconds.
```

并发循环的方式要更快些，这和我们预想的相吻合。这里概要说明了Scala中的一个有趣的问题。让我们回顾一下所学到的东西吧。

5.4.5　第三天我们学到了什么

第三天的学习内容不多，不过在强度上有了提高。我们构建了几个不同的并发程序，进行了直接的XML处理，将消息分发传递给actor，了解了模式匹配和正则表达式。

通过本章的学习，我们学到了4种彼此互相依赖的基本结构。首先，我们学会了如何使用Scala直接处理XML。我们可以使用一种类XQuery的语法查询单个元素信息或其属性信息。

然后，我们介绍了Scala版本的模式匹配。起初，它看起来像一个简单的`case`语句，不过介绍了哨兵、类型和正则表达式后，它们强大的功能就很快表现了出来。

接下来，转到并发的话题上来。我们使用了actor的概念，actor是为并发而构建出来的对象。它们通常拥有一个包含`react`或`receive`方法的循环，用于接收发给该对象的队列消息。最后，我们实现了一个内部模式匹配。我们使用原始的类作为消息，它们小巧、轻便、健壮且易于操作。如果需要在消息中增加参数，只需在类定义中添加属性即可，就像在sizer应用程序示例中加入URL那样。

和本书中介绍的其他语言一样，Scala远远比你在这里看到的更加强大。与Java类的交互也远远超出了我在这里向你所展示的内容。对于一些诸如柯里化的复杂概念，我们仅仅是浅尝辄止。不过你已经有了很好的基础，你应该继续深入学习下去。

5.4.6　第三天自习

现在你已经看到了Scala提供的一些高级特性了。你可以尝试自己去摆弄一下Scala。和以往一样，这些练习要求更高。

找

- 对于sizer程序，如果你没有为每个要跟踪的链接创建一个新的actor，这段程序的性能会发生怎样的变化？

做

- 修改sizer程序，增加一个计算页面上链接总和的消息。
- 加分题：让sizer跟踪给定页面上的所有链接并加载它们。例如，对给定页面"google.com"，sizer会计算出Google主页及主页所链接的页面的数量总和。

5.5 趁热打铁

到目前为止，我们对Scala语言的介绍比起其他语言来说更为详尽，这是因为Scala支持两种编程范型。面向对象的特性将Scala牢牢地定位成Java的替代品。与Ruby和Io不同，Scala使用的是静态类型策略。在语法上Scala借鉴了许多Java中的元素，包括大括号和构造器使用方法。

Scala对函数式编程概念和不变量提供了强大的支持。这门语言十分关注并发程序和XML，非常适合当前使用Java语言实现的大量种类繁多的企业应用。

Scala的函数式编程能力远远超出我在这一章中所涉及的内容。我没有介绍的结构包括柯里化、全闭包、多参数列表和异常处理。但是它们都是很重要的概念，可以增强Scala的功能和灵活性。

接下来看看Scala的核心优势和不足之处吧。

5.5.1 核心优势

Scala优势在于提供了一种高级的编程范型，将Java环境和一些精心设计的核心特性很好地整合在一起。特别是actor、模式匹配以及XML集成，它们都是十分重要且经过精心设计的功能。下面是各项优势及介绍。

1. 并发

Scala支持并发的方式代表了并行编程领域的一次重大进步。actor模型和线程池都是很受欢迎的改进，并且无需可变状态的并发应用设计能力也绝对是一个巨大的进步。

你曾在Io和现在的Scala中看到的actor模型很容易被开发人员理解，也广为学术界所研究。Java和Ruby都应该在这方面做些改善。

并发模型只是改进的一部分。当对象共享状态时，你必须争取使用不变值。Io和Scala至少部分做对了，它们允许可变状态，但是也提供了支持不变性的库和关键字。不变性是你能为改善并发代码设计能做的唯一的、最重要的事情。

最后，你在Scala中看到的消息传递语法与下一章节中Erlang十分相似。与Java的标准线程库相比，这是一个十分显著的改善。

2. 遗留Java的演化

Scala一开始就有着一个强大的、与生俱来的用户基础：Java社区。Scala应用可以直接使用Java库，并且必要时可以使用代理对象（proxy object）的代码生成功能，与Java的互操作性非常好。比古老的Java类型系统更先进的类型推断机制也是特别急需的。创立新的编程社区的最好方法就是充分接受现有的社区。Scala提供了一个更为简洁的Java，这方面做得很好，这种想法也是很有

价值的。

Scala也向Java社区贡献了许多新的特性。代码块成为了语言的一等类型构造，并且可以与核心集合库很好地集成在一起使用。Scala也以`trait`形式提供了一等类型mixin。模式匹配同样也是一个显著的改进。有了这许许多多的功能，Java程序员就可以拥有一门先进的语言，甚至无需学习更加高级的函数式编程范型。

加入函数式的结构，你的应用会得到显著的改进。Scala应用的总代码行数通常只是功能等同的Java应用的几分之一。这十分重要，良好的语言应该能够以更少的代码、最小的开销表达更复杂的想法。Scala兑现了这个承诺。

3. 领域特定语言

Scala灵活的语法和操作符重载使其成为了一门用来开发类Ruby风格领域特定语言的理想语言。记住，和Ruby一样，操作符只是简单的方法声明，大多数情况下你都可以重写它们。此外，可选的空格、句号以及分号让语法具有多种不同形式。再加上强大的mixin，这些都是一个DSL开发者努力寻找的工具。

4. XML

Scala提供内置的XML支持。模式匹配使得各种不同XML结构的解析块易于使用。将XPath语法集成到复杂的XML中使得代码变得更为简单、可读。这是很受欢迎的重要改进，特别是在大量使用XML的Java社区中。

5. 桥接

每种新出现的编程范型都需要一个桥梁。Scala具有成为这座桥梁的先天优势。函数式编程模型很重要，因为它可以很好地处理并发，并且处理器的并发程度也越来越高。Scala提供了一条迭代的路线以帮助程序员逐步实现目标。

5.5.2　不足之处

虽然我喜欢Scala的思想，不过我发现Scala的语法要求过多且偏学术性。虽说语法是有关个人品味的，但Scala语法负担确实比其他多数语言更重，至少对于我们这些老眼光的程序员是这样的。我也意识到了一些妥协削弱了Scala这样一个有效桥梁的价值。我只看到了三点不足，不过这些不足都是很重要的。

1. 静态类型

静态类型天生适合函数式编程语言，不过对于面向对象系统，Java风格的静态类型就是一场与魔鬼的交易。有时候由于必须满足编译器的需求，而这将给开发带来更多负担。静态类型带来的负担远远超出你的预期。对代码、语法以及程序设计的影响也是深远的。当我学会Scala时，我发现我自己一直处在一场语法和程序设计的持续战争中。`trait`稍微缓解了这个负担，让我找到了在程序员的灵活性和编译期检查需求之间的平衡点。

在本书后续章节中，你会通过Haskell看到一个纯函数式的强类型的静态类型系统是什么样子的。没有了这两种编程范型的负担后，类型系统将变得更加流畅和高效，并能更好地支持多态，而且在收益相同的情况下，对程序员的要求也没那么严格。

2. 语法

我真的发现Scala的语法有一些学术味道，并且看起来很吃力。是否将这个话题放到书中我一直犹豫不决，因为语法毕竟是很主观的，但是一些语法元素确实是有些令人困惑。有时，Scala保留了Java的惯例，诸如构造器。你会使用`new Person`而不是`Person.new`。而其他一些时候，Scala会引入一个新惯例，比如参数类型。在Java中，你会使用`setName(String name)`，而Scala则使用`setName(name: String)`。返回类型从Java方法声明的开头处挪到了结尾处。这些微小的差异让我一直不得不关注语法而不是代码逻辑。这种在Scala和Java间来回切换的问题会比原本耗费更多精力。

3. 可变性

当你构造一门充当桥梁角色的语言时，必须将妥协作为重要因素包含进去。Scala的一个重要的妥协就是引入了可变性。有了`var`，Scala就好似以某种方式打开了潘多拉的魔盒，因为可变状态可能导致各种各样的并发bug。不过如果你想将山丘上屋子中的那个特殊男孩儿带回家，那么这个妥协就是不可避免的。

5.5.3 最后思考

总之，我的Scala体验是喜忧参半。静态类型让我困惑，但同时我内心的Java程序员情节又让我十分感激改进的并发模型、类型推断机制以及内置的XML支持。Scala代表了编程艺术境界上的一次跃进。

5

如果我曾经在Java程序上有过较大投资，那我就会使用Scala来提升我的生产力。如果一个程序有着重要的可扩展性需求，需要使用并发时，我也会考虑使用Scala。商业上，这个科学怪人拥有了一个良好的契机，因为它代表了一座桥梁，并且完全包容了一个重要的编程社区。

第6章 Erlang

> 你听到了吗？安德森先生。这是命运的轰隆声。
>
> ——特工 Smith

似Erlang这般充满神秘感的语言寥寥无几。这门并发语言既可将难事化易，也可将易事变难。在健壮企业部署方面，它的虚拟机BEAM是唯一堪与Java虚拟机匹敌的对手。它调用起来十分高效，甚至效率以外的东西它都很少考虑。因此，它的语法也不像Ruby那样优雅和简洁。想想《黑客帝国》[1]里的特工Smith吧。

《黑客帝国》是1999年的一部经典科幻电影。它把你我现今身处的世界描绘成一个由计算机创造和维护、如同幻象一般的虚拟世界。特工Smith是这世界中的一段AI程序。他拥有化身为任意形式的惊人能力，也具有瞬间改变多处地点物理规律的力量。在他面前，你无处可逃。

6.1 Erlang 简介

Erlang其名，乍听之下很怪。但你若知道，它既是Ericsson Language的缩写，又恰是一位丹麦数学家的大名，你就不会再抱怨"这什么破名儿"了。作为电话网络分析的数学奠基人，Agner Karup Erlang[2]可称得上是赫赫有名。

1986年，Joe Armstrong在爱立信公司（Ericsson）开发了Erlang语言的首个版本。随后的五年间，Erlang在他的精心雕琢下日渐完善。20世纪90年代整整十年间，Erlang的发展都不温不火、时断时续，但到了2000年之后，它却开始成为众人瞩目的焦点。两个广受欢迎的云数据库CouchDB

① *The Matrix*，DVD版，导演：Andy Wachowski和Lana Wachowski（1999年）。发行商：加利福尼亚州伯班克市的华纳影业家庭娱乐公司（2007年）。（译者注：这部电影中译名为《黑客帝国》，是一部带有一定哲学意味的电影，大意是说我们所见的世界并非真正的现实世界，而是存在于母体Matrix当中、通过程序虚拟出来的世界。主人公Neo和同伴发现了母体背后的秘密，试图反抗母体的控制，也因此招致特工程序们的追杀。特工程序实际上是母体这个庞大的程序世界中的杀毒软件，他们被母体赋予了超凡的能力：可通过改写人类程序而不断借用别人的身体，虽具人类外表但并非人类；善于操控母体中的物理现象，可以扭曲万有引力、摩擦力等。拥有这些能力的特工是发现真相的人类所无法匹敌的。其中为首的一名特工就是Smith。最后Neo在爱情力量的感召下，唤醒了其"救世主"身份所拥有的重写母体程序的能力。他用这种能力修改了特工Smith的代码，让其消失得无影无踪。）

② Agner Karup Erlang（1878～1929），丹麦数学家、统计学家、工程师，排队论和电信流量工程学（Teletraffic Engineering）的创始人，一手开创了电话网络分析这一与我们日常生活息息相关的领域。

和SimpleDB，都是用Erlang开发出来的，此外，Erlang还是Facebook的聊天系统所采用的语言。正因为Erlang身怀可伸缩并发性和可靠性这两项拿手绝技，而其他语言在这两方面都力不从心，所以Erlang开始越来越多地成为人们谈论的话题。

6.1.1 为并发量身打造

Erlang是爱立信公司历经多年研究的产物，用来在电信领域中开发准实时[①]（near-real-time）且容错性较强的分布式应用。这类系统通常不可因维护而停止，因此软件开发费用极其高昂。20世纪80年代，爱立信对很多编程语言进行了研究，发现它们出于各种原因，均无法满足他们公司的需求。正是这些难以满足的需求，最终导致了一门全新语言的问世。

Erlang是一门函数式语言，可靠性方面的特性很多，可用于开发可靠性要求极高的系统。Erlang在替换模块时不必停止运行，这样就能边运行边维护电话交换机等设备。有些使用Erlang的系统已持续运行多年，从未因维护而中止过。可话说回来，要说到Erlang最关键的功能，那还得是并发。

哪些方法最适用于并发？在这一点上，并发领域的专家们有时意见并不统一。其中常见的一个争议是：线程更好还是进程更好？一个进程由多个线程组成，进程有自己的资源，而线程虽有自己的执行路径，但在同一进程内，各线程是资源共享的。尽管实现各异，但一般来说，线程比进程更轻量级。

1. 无线程

很多语言都采用线程实现并发，比如Java和C语言。线程占用资源较少，所以理论上说，使用线程可获得更优异的性能。线程的缺点，在于资源共享可能导致复杂而有缺陷的实现，而且这种资源共享必须用并发锁来管理，这也会产生性能瓶颈。为了在共享资源的两应用间分配控制权，线程机制需要借助信号量或是操作系统级别的锁。然而，Erlang另辟蹊径，尝试让进程也尽可能轻量级一些。

2. 轻量级进程

Erlang奉行的哲学是轻量级进程，这使它摆脱了在共享资源和性能瓶颈的泥沼中艰难跋涉的困境。Erlang的发明者煞费苦心地简化了应用程序中多进程创建、管理和通信的过程。分布式消息传递成为基本的语言结构，因此锁机制不再必要，并发性能也大有长进。

和Io一样，Erlang也将actor用在了并发当中，因此，消息传递就成为至关重要的概念。你可以在Erlang中依稀辨认出Scala的消息传递语法，因为它们的消息传递语法非常相似。Scala的actor代表一个对象，由线程池创建和维护，而Erlang的actor代表了一个轻量级进程。Erlang的actor从队列中读取外部进入的消息，并用模式匹配决定其处理方式。

3. 可靠性

Erlang虽然也有常规错误检测手段，但在容错应用中，需要处理的错误加起来远比传统应用

① 准实时是一个常用于电信和计算科学领域的术语，指的是由网络传输或数据处理所引起的较短的时间延迟。实际上，脱离具体标准谈论实时与准实时是没有意义的。很多情况下，准时实与实时并无明显差别；但在某些应用（如数据库整合）中，准实时的时间延迟可达到15~20分钟之久。

要多，这是常规手段无法解决的。Erlang解决这一问题的秘诀是“就让它崩溃”。由于Erlang能轻易监测到崩溃进程，因此终止相关进程并启动新进程也就不在话下了。

此外，Erlang还能做到“热插拔”代码。也就是说，你不必中止代码运行，就可以替换应用程序的各个部分。相比于其他同类分布式应用，这项功能将带给你更简单的维护策略。Erlang将健壮的“就让它崩溃”错误处理策略、“热插拔”以及创建开销极小的轻量级进程等优点集于一身，因此应用程序一次就能运行好几年都不宕机。

Erlang有这么多并发方面的传奇特性，实在是令人欲罢不能。它有三个最基本的要素：消息传递、进程创建和进程监控。用它新创建的进程是轻量级的，因此不必担心其控制区域内的资源可能受限。Erlang不仅可以尽可能地消除代码中的副作用和可变性，而且还可以很轻松地监测崩溃进程。有了这些特性，说它人见人爱真是一点都不过分。

6.1.2 Joe Armstrong博士访谈录

写作本书的过程中，我有幸接触到几位我最崇敬的人物，至少是通过电子邮件接触。Joe Armstrong博士是Erlang的发明者，也是《Erlang程序设计》[①]一书的作者。在我个人的英雄榜上，他排名很高。经过几次来往，我和这位来自瑞典斯德哥尔摩的Erlang语言首位实现者的访谈记录如下。

Bruce：你为什么要开发Erlang?

Armstrong博士：纯属巧合。我本来没打算发明一门新的编程语言。当时，我想找一种更好的方式来编写电信交换控制软件。我先试了试Prolog。Prolog是一门绝妙的语言，但它无法完全满足我的需要，既然如此，我就开始瞎倒腾Prolog。我琢磨着：“如果改变一下Prolog的编程方式，那会怎样？”于是，我写了个Prolog的元解释器，给它加上了并行进程，还加上了错误处理机制，诸如此类。就这样，过了一段时间，我给这些新增加的变化起了个名字——Erlang，一门新语言就这么诞生了。之后，越来越多的人加入这个项目，这门语言也逐渐发展起来。我们想出了编译它的方法，加入了更多东西，获得了更多用户……

Bruce：你最喜欢它哪一点呢?

Armstrong博士：我最喜欢它的错误处理、运行时代码升级机制，还有位级（bit-level）模式匹配。错误处理是这门语言最不为人所知的部分，也是与其他语言差别最大的部分。Erlang的“非防御”编程和“就让它崩溃”这一套概念，既是它的独门绝学，也是它与传统方法截然相反之处。不过，这样做的确能编出简洁而漂亮的程序。

Bruce：如果能让时光倒流，你最想改变哪项特性?（换言之，你也可以回答这样一个问题：Erlang最大的局限是什么？）

Armstrong博士：这问题很难，我可能会在不同时间给出不同答案。为这门语言添加一些移动特性应该不错，这样我们就能通过移动通信网络传送计算结果。我们可以用库代码来做这件事，

① 英文名为*Programming Erlang: Software for a Concurrent World*[Arm07]。中译本于2008年由人民邮电出版社出版。

但它并不被语言本身所支持。我现在想，如果追本溯源，把Prolog式的谓词逻辑加入Erlang，产生一种谓词逻辑和消息传递的全新组合，那想必会十分美妙。

还有不少小改动也是我想做的，比如说，加入散列映射、高阶模块，等等。

要是推倒重来，我可能会更多地把心思花在各项编程事务的协调上，比如说，如何运作有大量代码的大型编程项目——如何管理代码版本、如何搜索想要的东西、各种事物如何演化。当程序员编写了大量代码之后，他的任务就不再是编写新代码，而是准确找到现有代码，并把现有代码整合起来。因此，搜索和协调就变得日渐重要。如果把GIT和Mercurial[①]这类系统的思想吸收到Erlang之中，再给它加上类型系统，使它能在可控条件下理解代码是如何演化的，那我想应该会带来不错的效果。

Bruce：在实际产品中，你见过的最特别的Erlang应用是什么？

Armstrong博士：嗯，其实我并不会太过惊讶，因为我早就知道它能达到何等高度。当我把Ubuntu版本升级到Karmic Koala[②]时，我发现，它为了支持正在我机器上运行的CouchDB，而在后台悄悄启动了Erlang。这就好比Erlang在雷达的严密监控之下，偷偷溜进了数千万用户的计算机当中。

本章中，我们一开始会介绍一些Erlang的基础知识。之后，我们会深入研究Erlang作为一门函数式语言的各种特性。最后，我们还要花点时间看看Erlang的并发性特性，以及它超酷的可靠性特性。没错，朋友，可靠性也能这么赞。

6.2　第一天：以常人面目出现

特工Smith是被称作“母体”（Matrix）的虚拟现实世界中的程序，只要其他程序或虚拟人扰乱母体的正常运行，他就会将他们杀死。但他也拥有一项常使自己陷于险境的能力，就是以常人面目出现。同样，我们在本节当中，也要先看看Erlang编写通用应用程序的能力。我将尽我所能，让你看看“常态”下的Erlang是什么样的。想做到这一点并非易事。

6

如果你在读这本书之前，仅用面向对象语言编过程，那你或许会感到些许不适。不过，你不必太过在意这种不适。前面你见识过Ruby的代码块、Io的actor、Prolog的模式匹配，还有Scala的分布式消息传递。这些在Erlang中也都是最基本的思想，但本章会选择从另一个重要思想切入主题。Erlang是本书出现的第一门函数式语言。（Scala是函数式/面向对象混合语言。）对你而言，这意味着以下几点：

- 程序将完全基于函数编写出来，压根儿就没有对象这种东西；
- 通常来说，给定相同输入，这些函数将返回相同的值；
- 这些函数通常没有副作用，也就是说，它们不改变程序的状态；
- 任何变量都只能赋值一次。

第一条规则还算好对付，但再加上后面那三条，你就只有目瞪口呆的份儿了。至少有那么一

① GIT和Mercurial都是分布式版本控制系统。

② Ubuntu 9.10版的代号。

段时间，你都会保持这种目瞪口呆的状态。不过你要知道，你完全能够学会这种编程方式，而且以这种方式编出来的程序，正是为并发量身打造的。当你摆脱了程序的可变状态，并发就会变得出奇简单。

如果你仔细看过这几条规则，就会发现，第二条和第三条都带有*通常*这个词。Erlang并非纯函数式语言，它允许出现少数违反规则的情况。在本书介绍的语言中，Haskell是唯一的一门纯函数式语言。尽管如此，你用Erlang编程时，还是能感受到浓郁的函数式编程风格，而且大多数情况下，你都要遵守上面提到的四条规则。

6.2.1　新手上路

我用的Erlang版本是R13B02,但本章涉及的基本知识应该能在任何版本的Erlang中正常运行。你可以输入`erl`（某些Windows系统是`werl`）来打开Erlang的命令行，像下面这样：

```
batate$ erl
Erlang (BEAM) emulator version 5.4.13 [source]

Eshell V5.4.13  (abort with ^G)
1>
```

这章前面的大部分任务都能用命令行完成，就像其他各章所做的那样。和Java一样，Erlang也是编译型语言。它可以用`c(`*文件名*`).`（结尾要加上句点“.”）编译文件。按Ctrl+C可以退出命令行或跳出循环。下面，我们开始学习基本功。

6.2.2　注释、变量和表达式

我们先来学一些基本语法。打开命令行，输入下列代码：

```
1> % This is a comment
```

很简单。注释以%符号开头，从该符号开始、直到该行末尾所包含的内容，都会被Erlang当作注释。对注释作语法分析时，Erlang会将整条注释转换为一个空格。

```
1> 2 + 2.
4
2> 2 + 2.0.
4.0
3> "string".
"string"
```

每个语句都必须以句号结尾。上面给出了一些Erlang的基本类型：字符串、整数、浮点数。下面来看看列表：

```
4> [1, 2, 3].
[1,2,3]
```

如同Prolog家族的其他语言一样，列表是用方括号表示的。这里有个让人略微感到奇怪的地方：

```
4> [72, 97, 32, 72, 97, 32, 72, 97].
"Ha Ha Ha"
```

也就是说，字符串其实是个列表。这就好像特工Smith冲着你母亲哈哈大笑，哎，真粗鲁[①]。前面的2+2.0这行代码，说明Erlang可以做一些基本类型强制转换。现在我们用不当的类型转换，让代码报错：

```
5> 4 + "string".
** exception error: bad argument in an arithmetic expression
 in operator  +/2
    called as 4 + "string"
```

和Scala不同，Erlang不能在字符串和整数间强制转换。下面为变量赋值：

```
6> variable = 4.
** exception error: no match of right hand side value 4
```

糟了。这个错误告诉你，用黑客帝国中的特工比喻Erlang并不是那么恰当。有时，这门讨厌的语言真的是有脑无心。这条错误消息说的是Erlang的模式匹配。它之所以出错，是因为这里的variable其实是个原子。变量的话，必须以大写字母开头才行。

```
7> Var = 1.
1
8> Var = 2.

=ERROR REPORT==== 8-Jan-2010::11:47:46 ===
Error in process <0.39.0> with exit value: {{badmatch,2},[{erl_eval,expr,3}]}

** exited: {{badmatch,2},[{erl_eval,expr,3}]} **
8> Var.
1
```

如上述代码所示，变量以大写字母开头，且它们是不可变的，你只能为每个变量赋值一次。这一思想让初次接触函数式语言的大多数程序员纠结不已。下面，我们介绍几个复杂一点的数据结构。

6

6.2.3 原子、列表和元组

函数式语言中，符号（symbol）的作用更为突出。它们是最基本的数据元素，可以表示任何你想为之起名的事物。在本书介绍的其他各门语言中，你曾见过符号。但在Erlang中，符号叫做原子，并以小写字母开头，它们用来表达很小的事物粒度。你可以像下面这样使用它们：

```
9> red.
red
10> Pill = blue.
blue
11> Pill.
blue
```

① 本书把Erlang比作特工Smith，因此作者想借特工Smith的粗鲁举动，说明Erlang并不那么友好，可能会产生用户难以理解的代码。用列表表示字符串的代码即为一例。

这里的red和blue都是原子。这些原子起什么名都行，用来符号化现实世界的事物。在代码中，我们先是返回了一个简单原子red。然后，我们把一个名为blue的原子赋给了变量Pill。当原子和更强大的数据结构结合起来时，原子会变得更有趣，我们在后面会讲到。现在，我们用前面学过的基本类型构建列表。列表用方括号表示：

```
13> [1, 2, 3].
[1,2,3]
14> [1, 2, "three"].
[1,2,"three"]
15> List = [1, 2, 3].
[1,2,3]
```

Erlang的列表语法看着挺眼熟。列表是异质[1]和变长的。我们可以把列表赋给变量，就像给变量赋一个基本类型的值一样。而元组是定长的异质列表：

```
18> {one, two, three}.
{one,two,three}
19> Origin = {0, 0}.
{0,0}
```

这段代码没什么特别。你也许看得出，Erlang受Prolog影响颇深。稍后我们会讲到模式匹配，那时，你将看到元组长度在匹配元组时所起的作用。你不能用三元组去匹配二元组。但在匹配列表时，长度是可以变化的，就像Prolog所做的那样。

在Ruby中，可以用散列表，把值关联到名字上。而在Erlang中，如果想用映射或散列的话，常常会用到元组：

```
20> {name, "Spaceman Spiff"}.
{name,"Spaceman Spiff"}
21> {comic_strip, {name, "Calvin and Hobbes"}, {character, "Spaceman Spiff"}}.
{comic_strip,{name,"Calvin and Hobbes"},
          {character,"Spaceman Spiff"}}
```

上面的代码中，comic_strip（四格漫画）是用散列的形式表示出来的。我们以原子为键、字符串为值。你还可以混合使用元组和列表，比如在列表里用元组表示漫画。如果用散列的话，如何访问其中的各个元素呢？假如此刻，你脑中蹦出了Prolog这个词，那么恭喜你，你找到了正确答案。利用模式匹配，就可以访问那些元素。

6.2.4　模式匹配

如果吃透了Prolog那章，模式匹配也就打下了扎实的基础。不过必须指出的是，Erlang和Prolog在模式匹配上有一处重大不同。Prolog中，你定义一条规则，会匹配数据库上的所有的值。也就是说，Prolog会将模式匹配作用于所有条目。Erlang的运作方式则类似于Scala，一个匹配仅仅作用于单个值。现在，我们就来看看模式匹配是如何从元组中把值取出来的。例如，我们有这样一个表示人的元组Person：

① 即可包含不同类型元素。

```
24> Person = {person, {name, "Agent Smith"}, {profession, "Killing programs"}}.
                {person,{name,"Agent Smith"},
                        {profession,"Killing programs"}}
```

我们想把元组中的name赋值给变量Name，把profession赋值给Profession。使用下面的匹配，我们能巧妙地做到这点：

```
25> {person, {name, Name}, {profession, Profession}} = Person.
{person,{name,"Agent Smith"},
        {profession,"Killing programs"}}
26> Name.
"Agent Smith"
27> Profession.
"Killing programs"
```

Erlang会把两个数据结构匹配起来，并将元组中的那些值赋给变量。原子匹配其自身，因此，上面的代码只不过是把变量Name和"Agent Smith"匹配起来、变量Profession和"Killing programs"匹配起来而已。这一特性的工作原理和Prolog十分相似，同时，它也是常用的基本判断结构。

若是你已经习惯了Ruby或Java风格的散列表，那么散列表开头加个person原子看来或许有几分古怪。但在Erlang中，我们常常要用到多匹配语句和多类型元组。照刚才说的那样设计数据结构，Erlang就不必理会开头原子之外的部分，而能快速匹配所有用来表示人的元组。

列表的模式匹配也很像Prolog：

```
28> [Head | Tail] = [1, 2, 3].
[1,2,3]
29> Head.
1
30> Tail.
[2,3]
```


是不是易如反掌？我们还可以把列表头绑定到几个变量上：

```
32> [One, Two|Rest] = [1, 2, 3].
[1,2,3]
33> One.
1
34> Two.
2
35> Rest.
[3]
```

如果列表没有足够元素，则模式不匹配：

```
36> [X|Rest] = [].
** exception error: no match of right hand side value []
```

这样一来，另一类错误消息也就说得通了：假如忘了把变量首字母大写，会看到下面的错误消息：

```
31> one = 1.
** exception error: no match of right hand side value 1
```

如前所述，=语句不是简单的赋值语句，而是模式匹配。你要求Erlang用原子one去匹配整数1，可它做不到这点。

位匹配

有时，你需要在位级别上存取数据。如果想在较小的空间内塞下较多数据，或想处理JPEG或MPEG这样的预定义数据，那么每一个位放了什么就特别重要。Erlang能让你轻松地把几个数据片段打包到一个字节当中。做到这一点，需要两种操作：打包和解包。在Erlang中，位图与其他集合（collection）的工作方式毫无二致。为了打包一个数据结构，你要让Erlang知道每个元素各占多少位，像下面这样：

```
1> W = 1.
1
2> X = 2.
2
3>
3> Y = 3.
3
4> Z = 4.
4
5> All = <<W:3, X:3, Y:5, Z:5>>.
<<"(d">>
```

<<和>>把二进制模式包含到构造器中。在上面的例子中，其构造器表示变量W占3位、X占3位、Y占5位、Z占5位。既然有打包，当然也要能解包才行。你也许已猜到解包的语法：

```
6> <<A:3, B:3, C:5, D:5>> = All.
<<"(d">>
7> A
1
8> D.
4
```

这里的语法和元组、列表一模一样，剩下的事全交给模式匹配打理就行。有了这些位操作，Erlang在执行底层任务时就会变得异常强大。

这一章所有较重要的概念都已介绍完毕，我们没用多大工夫就学到了大量的基础知识。不管你信不信，我们现在几乎完成了Erlang第一天的学习。不过，我们还要介绍一个最重要的概念——函数。

6.2.5　函数

和Scala不同，Erlang是动态类型的。你不必担心数据元素的赋值类型是什么。Erlang和Ruby一样，都是动态类型的。它会在运行时根据引号或小数点等语法线索绑定类型。我现在要打开一个新的命令行。不过在此之前，我想先介绍几个新概念。我们会在后缀为.erl的文件里编写函

数。这个文件包含了用于模块的代码，必须经过编译才能运行。编译后，它会产生一个.beam可执行文件。这个.beam已编译模块，将运行在名为beam的虚拟机之上。

铺垫已做足，该到写几个简单函数的时候了。

创建一个下面这样的文件：

erlang/basic.erl

```
-module(basic).
-export([mirror/1]).

mirror(Anything) -> Anything.
```

第一行定义了模块名称，第二行定义想要在模块外部执行的函数。该函数的名字是mirror，/1表示它带有一个参数。最后一行是真正要执行的函数内容，可以看出，它受Prolog风格的影响不小。这一行先是指定了要定义的函数名mirror，并确定了传入参数Anything。参数之后，紧跟着一个->符号，简单返回函数的第一个参数。

以上就是一个完整的函数定义。我们可以在代码文件的所在目录中打开命令行，并像下面这样编译代码：

```
4> c(basic).
{ok,basic}
```

这样，就编译了basic.erl，你会在该目录中看到basic.beam文件。你可以像下面这样运行它：

```
5> mirror(smiling_mug).
** exception error: undefined shell command mirror/1
6> basic:mirror(smiling_mug).
smiling_mug
6> basic:mirror(1).
1
```

注意，只有函数名是不行的，还得在前面加上模块名，再跟上一个冒号。函数执行起来非常简单。

还有一件事需要注意：我们能把函数的参数Anything绑定到两个不同类型上①。Erlang是动态类型语言，这正合我意。经历过Scala的强类型，我就像在西伯利亚过了个周末，或者退一步说，在皮奥里亚②过了个周末之后，终于回到了家。

现在看看稍微复杂一些的函数是什么样的。下面的函数定义了几种匹配选择。

创建一个matching_function.erl文件：

erlang/matching_function.erl

```
-module(matching_function).
-export([number/1]).
```

① 前面的示例代码中，函数参数就绑定了原子和整数两个不同的类型。

② 美国伊利诺伊州的一座城市，冬季颇为寒冷。

```
number(one)   -> 1;
number(two)   -> 2;
number(three) -> 3.
```

然后，像下面这样执行它：

```
8> c(matching_function).
{ok,matching_function}
9> matching_function:number(one).
1
10> matching_function:number(two).
2
11> matching_function:number(three).
3
12> matching_function:number(four).
** exception error: no function clause matching matching_function:number(four)
```

这是我介绍的第一个多匹配选择的函数。每种可能的匹配都有函数名、匹配参数以及“->”符号后的执行代码。每种匹配情况下，都仅仅让Erlang返回一个整数。最后一条匹配语句以“.”结尾，其他匹配语句用以“;”结尾。

就像在Io、Scala和Prolog里一样，递归扮演着非常重要的角色。Erlang和Prolog都对尾递归进行了优化。下面是阶乘函数的实现：

erlang/yet_again.erl

```
-module(yet_again).
-export([another_factorial/1]).
-export([another_fib/1]).

another_factorial(0) -> 1;
another_factorial(N) -> N * another_factorial(N-1).

another_fib(0) -> 1;
another_fib(1) -> 1;
another_fib(N) -> another_fib(N-1) + another_fib(N-2).
```

和其他各种阶乘实现一样，这里的阶乘也是递归定义的。既然有阶乘，自然也少不了斐波那契数列。

试着让代码以下列形式运行。这会花一点时间：

```
18> c(yet_again).
{ok,yet_again}
19> yet_again:another_factorial(3).
6
20> yet_again:another_factorial(20).
2432902008176640000
21> yet_again:another_factorial(200).
788657867364790503552363213932185062295135977687173263294742533244359
449963403342920304284011984623904177212138919638830257642790242637105
061926624952829931113462857270763317237396988943922445621451664240254
033291864131227428294853277524242407573903240321257405579568660226031
```

```
90417032406235170085879617892222278962370389737472000000000000000000
0000000000000000000000000000
22> yet_again:another_factorial(2000).
3316275092450633241175393380576324038281117208105780394571935437060380
7790560082240027323085973259225540235294122583410925808481741529379613
1386633526343688905634058556163940605117252571870647856393544045405243
9574670376741087229704346841583437524315808775336451274879954368592474
... and on and on...
0000000000000000000000000000000000000000000000000000000000000000000000
```

太棒了！这真是与众不同。现在，你见识到把Erlang比作特工Smith的绝妙之处了吧。如果你没时间运行代码，我向你保证，得到这结果绝对只有一眨眼工夫。我不知道Erlang能表达的最大整数是多少，但我想悄悄告诉你，对我来说这肯定是足够了。

这是一个不错的起点。你创建了一些简单的函数，也看到了它们的运行情况。是时候温习一下第一天的学习成果了。

6.2.6 第一天我们学到了什么

Erlang是一门函数式语言。它是强类型和动态类型语言。它没有太多语法，而且看起来和典型的面向对象语言完全不同。

和Prolog一样，Erlang没有对象的概念。它和Prolog有相当紧密的联系。Erlang的模式匹配结构和多函数入口点对你来说很熟悉，你可以利用它们通过递归解决一些问题。Erlang这门函数式语言没有可变状态的概念，甚至没有副作用的概念。维护程序状态让人头疼，但你可以从Erlang中学到一系列新技巧。不过，你也将很快看到事情的另外一面，消除状态和副作用将对管理并发的方式产生极其重大的影响。

在这第一天中，你用命令行和编译器两种方式运行了代码。首先，我们把重点放在基础知识上。学会了一些基本表达式，还写了一些简单的函数。和Prolog一样，Erlang的函数也具有多个入口点。这一天也用到了基本的模式匹配。

6

介绍了基本的元组和列表。元组取代了像Ruby那样的散列表，构成了Erlang数据结构的基础。学习了列表和元组的模式匹配。这些思想能让你迅速把行为加到元组或进程间消息上，我们在后面的章节会讲到这点。

第二天，我们将进一步介绍这些基本的函数式思想。我们将学习如何编写在并发世界良好运行的代码，但第二天结束之时，我们仍到不了完全掌握并发编程的境界。现在，花点时间做做习题，实践一下现在已经学到的知识。

6.2.7 第一天自习

Erlang的在线社区成长得十分迅速。在旧金山举办的会议影响力也越来越大。与Io和C语言不同，用Google搜Erlang就能找到你需要的东西。

找

- Erlang语言的官方网站。

- Erlang函数库的官方文档。
- Erlang OTP库的官方文档。

做

- 写一个函数，用递归返回字符串中的单词数。
- 写一个递归计数到10的函数。
- 写一个函数，在给定输入为{error, Message}或success的条件下，利用匹配相应地打印出"success"或"error: message"。

6.3　第二天：改变结构

本节中，我们将进一步了解特工Smith的能力。《黑客帝国》中的特工们都有着超人般的威力。他们能轻巧地闪避子弹，也能一拳把水泥墙击穿。而函数式语言，它们处在比面向对象语言更高的抽象层次上，虽然理解起来更有难度，但也因此能仅用几行代码，就表达相当丰富的含义。

特工Smith还能够化身为在这个母体中生活的任何一个人的模样。这也是函数式语言的一项重要能力。你将学会如何在列表上应用函数，使这列表瞬间变化为你想要的形式。想把购买商品列表变成价格列表吗？还是把URL列表变为包含内容和URL的元组？在函数式语言面前，这些问题都是小菜一碟。

6.3.1　控制结构

我们先从Erlang较为平淡无奇的部分——基本控制结构开始学起。你将注意到，这一节内容比Scala里的相应内容要短得多。你会经常看到充斥着case语句的程序，这是因为，编写并发程序时，case语句可以表示即将处理哪一条消息。相比之下，if语句用的就少得多了。

1. case

我们这就开始学习case语句。大多数时候，我们都是为了调用函数而想到模式匹配的。同样，我们可以把case这种控制结构想象成可随时随地使用的模式匹配。举个例子，有一个变量Animal，你想根据它的值来执行特定代码：

```
1> Animal = "dog".
2> case Animal of
2>     "dog" -> underdog;
2>     "cat" -> thundercat
2> end.
underdog
```

这个例子中，字符串匹配了第一个子句，并返回原子underdog。和Prolog一样，你可以用下划线（_）来匹配任意符号，像下面这样（注意，变量Animal的值仍然是"dog"）：

```
3> case Animal of
3>     "elephant" -> dumbo;
3>     _ -> something_else
3> end.
something_else
```

这里Animal的值不是"elephant"，所以它匹配了后一个子句。你也可以在其他任意一个Erlang匹配中使用下划线。不过我想指出，这里存在的一个基本语法缺陷。注意，除了最后一个子句之外，所有case子句都以分号结尾。这就意味着，如果想调换一下子句顺序，也要相应调整子句结尾处的分号才行。其实对Erlang来说，在最后一个子句后加上可选的分号一点儿都不难。不错，这种语法形式有其逻辑性：以分号作为case子句之间的分隔符。但它用起来就是非常不爽。这就好像特工Smith把沙子踢起来，弄得你小侄子满头满脸都是。我仿佛又听到了他得意的笑声[①]。然而，如果他想评为当月最佳特工，那他现有的这些技能还不太够。接下来看看另一项基本技能——if。

2. if

case语句用的是模式匹配，而if语句用的是"哨兵"。Erlang中的哨兵是指成功匹配所必须满足的条件。稍后，我们将介绍模式匹配中使用的哨兵，但哨兵最主要的用途还是在if语句中。先以if关键字开头，后面跟几个"哨兵→表达式"子句为例，像下面这样：

```
if
  ProgramsTerminated > 0 ->
    success;
  ProgramsTerminated < 0 ->
    error
end.
```

如果没匹配上，会发生什么呢？

```
8> X = 0.
0
9> if
9>   X > 0 -> positive;
9>   X < 0 -> negative
9> end.
** exception error: no true branch found when evaluating an if expression
```

与Ruby或Io不同，这里的子句必须有一个为真，因为if其实是个函数。此外，每一种匹配情况都必须有一个返回值。如果你确实想要case语句的效果，那可以令最后一个哨兵为true，像下面这样：

```
9> if
9>   X > 0 -> positive;
9>   X < 0 -> negative;
9>   true  -> zero
9> end.
```

控制结构的内容就这么多。从高阶函数和模式匹配当中，可以汲取更多有用的知识，所以我们不再执迷于控制语句，而向函数式语言更深的领域进军。下面将介绍高阶函数，并用它们处理列表。学会了这些函数，你将逐渐懂得解决更复杂问题的方法。

① 与前文"冲着你母亲哈哈大笑"的比喻一样，体现了Erlang的不友好。

6.3.2　匿名函数

你可能还记得，高阶函数有两种，一种是以函数为返回值，一种是以函数为参数。Ruby是拿代码块当作高阶函数来用，主要用法是把代码块传递给列表，让它在列表上迭代执行。而在Erlang中，你可以将任意函数赋给变量并传递它们，就跟其他数据类型一样。

前面你见过一些高阶函数相关的概念，但这里我们仍要打好Erlang高阶函数的基础，然后再构建较高层次的抽象。我们先从匿名函数学起。下面是把函数赋给变量的代码：

```
16> Negate = fun(I) -> -I end.
#Fun<erl_eval.6.13229925>
17> Negate(1).
-1
18> Negate(-1).
1
```

第16行用到了一个新的关键字`fun`，这个关键字定义了一个匿名函数。在上面的例子中，定义的匿名函数带一个参数`I`并返回`-I`。这个匿名函数被赋给了`Negate`。这里需要明确一点：`Negate`并不是函数返回的值，它就是这函数本身。

这个例子中蕴含着两个重要思想。第一，我们把函数赋给了变量。这样就能像传递其他数据一样传递行为；第二，我们可以轻松地在函数内部调用其他函数，只需在参数列表中指定函数即可。注意，Erlang是动态类型语言，你不必操心函数的返回类型，也可免于受到Scala那样的繁复语法之苦。这种函数用法的缺点是，函数可能出现执行错误。稍后我会给出Erlang弥补这一缺陷的几种方法。

这种全新的功能用处不小。我们将用它处理早在Ruby一章中就曾见到过的`each`、`map`、`inject`等概念。

6.3.3　列表和高阶函数

如你所见，列表和元组是函数式语言的核心。第一门函数式语言就是凭借列表起家的，它把一切事物都建立在列表基础之上，这绝非巧合。本节中，我会在列表上应用高阶函数。

1. 在列表上应用高阶函数

此时此刻，你应该很容易就能理解这一节的主要思想了。我们将用函数来帮我们管理列表。其中一些函数，比如`foreach`，将用来迭代列表；而另一些函数，比如`filter`或`map`，将通过筛选或映射到其他函数的方式返回列表。此外，还有其他一些处理列表的函数，比如`foldl`和`foldr`，它们采用的方式类似于Ruby的`inject`函数和Scala的`FoldLeft`函数等，即把结果汇总起来。打开一个新的命令行窗口，定义一两个列表，然后马上开始我们的实践。

首先，我们解决最基本的迭代。`lists:foreach`方法带有一个函数和一个列表，其中函数可以是匿名的，如下面的代码所示：

```
1> Numbers = [1, 2, 3, 4].
[1,2,3,4]
2> lists:foreach(fun(Number) -> io:format("~p~n", [Number]) end, Numbers).
1
2
3
4
ok
```

第2行代码在语法上有点难度，有必要把它梳理一下。首先，我们调用了一个名为lists:foreach的函数，它的第一个参数是匿名函数fun(Number) -> io:format("~p~n", [Number]) end。这个匿名函数带有一个参数，它利用io:format函数[①]，把传入函数的任意值打印出来。之后，foreach还带有第二个参数Numbers。在代码的第一行中，已给出过Numbers的值。如果把这匿名函数单拿出来写成一行，代码会清楚得多：

```
3> Print = fun(X) -> io:format("~p~n", [X]) end.
```

这里，变量Print绑定到了函数io:format上。于是，我们可以像下面这样简化代码：

```
8> lists:foreach(Print, Numbers).
1
2
3
4
ok
```

这就是最基本的迭代。现在，我们来看看映射函数。在Erlang中，映射函数采用的是与Ruby的collect函数类似的方式。它把列表的每个值传递给函数，并用返回结果创建一个列表。和lists:foreach一样，lists:map也带有一个函数参数和一个列表参数。下面，我们在前面给出的数字列表上应用map，使列表中的每个值都加1：

```
10> lists:map(fun(X) -> X + 1 end, Numbers).
[2,3,4,5]
```

这段代码非常简单。这里所用的匿名函数是fun(X) -> X + 1 end，它把列表的每个值都加了1。然后，lists:map用匿名函数返回的结果构建了一个列表。

映射函数的定义也是一目了然：

```
map(F, [H|T]) -> [F(H) | map(F, T)];
map(F, [])    -> [].
```

的确一目了然。这里，把F映射到列表上，就相当于F(head)后面跟着map(F,tail)。在稍后学习列表解析时，还将看到映射定义的一个更简洁的版本。

接下来，我们用布尔运算来筛选列表。定义一个匿名函数，并把它赋给Small：

① io:format中的~p~n类似于C语言的printf中的格式化字符串，~p将转换为后面给出的参数列表，~n是换行符，其后的[Number]为要输出的参数列表。——原书注

```
11> Small = fun(X) -> X < 3 end.
#Fun<erl_eval.6.13229925>
12> Small(4).
false
13> Small(1).
true
```

现在可以把这个函数作为参数，并利用它筛选列表中的元素。函数lists:filter将用所有满足Small的元素（小于3的元素）构建列表：

```
14> lists:filter(Small, Numbers).
[1,2]
```

可以看到，Erlang的这种编程方式是相当方便的。此外，我们还可以用Small函数结合all或any来测试列表。只有列表所有元素都满足筛选器（filter）时，lists:all才返回true，如下所示：

```
15> lists:all(Small, [0, 1, 2]).
true
16> lists:all(Small, [0, 1, 2, 3]).
false
```

而lists:any，是只要列表有一个元素满足筛选器，它就会返回true：

```
17> lists:any(Small, [0, 1, 2, 3]).
true
18> lists:any(Small, [3, 4, 5]).
false
```

如果把它们用在空列表上，返回结果如下所示：

```
19> lists:any(Small, []).
false
20> lists:all(Small, []).
true
```

你可能已经猜到了，all会返回true（表示列表所有元素都满足筛选器，尽管列表里一个元素都没有），而any会返回false（表示空列表中没有满足筛选器的元素）。在这两种情况下，不管用什么样的筛选器，它们的返回值都是这样。

你还可以用处在列表头位置的所有满足筛选器的元素组成一个列表，或者把处在列表头位置、满足筛选器的元素舍弃掉：

```
22> lists:takewhile(Small, Numbers).
[1,2]
23> lists:dropwhile(Small, Numbers).
[3,4]
24> lists:takewhile(Small, [1, 2, 1, 4, 1]).
[1,2,1]
25> lists:dropwhile(Small, [1, 2, 1, 4, 1]).
[4,1]
```

这些测试在完成某些任务时很有用，比如处理或舍弃消息头。下面，我们将学习foldl和foldr这两个函数。学过它们之后，我们也将结束这一天的高强度学习。

2. foldl

我想你之前已经看到过这些概念。如果你是Neo，母体的这部分知识你已了然于胸，那么只需了解一些简单的例子就可以继续战斗了。对某些人来说，只要稍微花点儿工夫就能掌握`foldl`。因此，我想用几种不同的方法讲解它。

记住，当我们想把某个函数遍历列表所得的结果汇总起来时，以下这些函数会非常有用。匿名函数的一个参数是累加器，另一个参数表示每次迭代的元素值[1]。`lists:foldl`带有一个函数参数、一个累加器的初始值参数以及一个列表：

```
28> Numbers.
[1,2,3,4]
29> lists:foldl(fun(X, Sum) -> X + Sum end, 0, Numbers).
10
```

为简化代码起见，我们把匿名函数拆分出来，放到一个变量当中，再选用一个合适的变量名，令我们的意图更加直截了当：

```
32> Adder = fun(ListItem, SumSoFar) -> ListItem + SumSoFar end.
#Fun<erl_eval.12.113037538>
33> InitialSum = 0.
0
34> lists:foldl(Adder, InitialSum, Numbers).
10
```

哈哈，这样看起来就清楚多了。从代码中可以看出，我们保存了列表和，而这个和是不断增加的。我们把`SumSoFar`以及列表`Numbers`中的每一个数传递给函数`Adder`，一次迭代只传递列表中的一个数。每一次迭代，其总和`SumSoFar`都会增大，而`lists:foldl`函数会记住这个不断变化的总和，并将其传回给`Adder`。最终，`lists:foldl`将返回最后一次迭代后得到的列表和。

到现在为止，你看到的都是在现有列表上运行的函数。我还没让你见识到每次只构建列表中的一部分该怎么做。下面，我们就来攻克列表构建的这部分内容。

6.3.4 列表的一些高级概念

我刚才介绍的所有列表概念，都不过是你在其他语言中已经见过的思想的扩展。可好戏还在后面，你马上就能学到一些更为精妙的概念。我们尚未涉及列表构建的内容，而且用到的那些简单代码块，也只是相当基本的一种抽象。

1. 列表构造

表面上看，在没有可变状态的情况下，列表构建似乎是一件困难的任务。如果用Ruby或Io这类语言，那可以把元素一个一个地加到列表中。但如果用的是没有可变状态的语言，我们也可以返回一个添加了新元素的列表。我们通常从列表头开始添加。这种添加或构造也要用到`[H|T]`结构，但不是在匹配的左边，而是在右边。下面的程序用列表构造方法，把列表中的每个元素都加倍：

---

① 第一个参数是迭代元素值，第二个参数是累加器。

erlang/double.erl

```
-module(double).
-export([double_all/1]).

double_all([]) -> [];
double_all([First|Rest]) -> [First + First|double_all(Rest)].
```

这个模块输出double_all的函数。该函数有两个不同的子句。第一个子句表示对空列表使用double_all，将仍然返回空列表。这个子句描述的规则使递归最终能够停止。

第二个子句规则用到了[H|T]结构。该结构不仅出现在匹配语句的谓词中，还出现在函数定义中。在匹配左边是[First|Rest]结构，用于把列表拆分成第一个元素和第一个元素之外的部分。

如果把它用在匹配右边，那么它实现的功能，就变成了是列表构造而不是分解。上面的例子中，[First + First|double_all(Rest)]即代表列表构造，其中First + First是列表的第一个元素，double_all(Rest)是列表的其余部分。

你可以像往常那样编译和运行程序：

```
8> c(double).
{ok,double}
9> double:double_all([1, 2, 3]).
[2,4,6]
```

下面在命令行中结合使用“|”，重新审视一下列表构建这一概念：

```
14> [1| [2, 3]].
[1,2,3]
15> [[2, 3] | 1].
[[2,3]|1]
16> [[] | [2, 3]].
[[],2,3]
17> [1 | []].
[1]
```

这结果并不出人意料。第二个参数必须是个列表，而左边的第一个参数将加到第二个参数之前，作为新列表的第一个元素。

下面，我们来看一个更高级的Erlang概念——列表解析。这个概念是我们前面讲过的一些概念的综合。

2. 列表解析

任何一门函数式语言当中，map都是最重要的函数之一。只要用上map，列表就可以自由变化，就像黑客帝国里的那些反派一样。由于这种特性的重要性，Erlang进一步提供了一种更加简明、可一次执行多个变换的形式。

我们现在重新打开一个命令行，开始这一节的学习。先用老方法来做映射：

```
1> Fibs = [1, 1, 2, 3, 5].
[1,1,2,3,5]
2> Double = fun(X) -> X * 2 end.
#Fun<erl_eval.6.13229925>
3> lists:map(Double, Fibs).
[2,2,4,6,10]
```

这里有一个数字列表Fibs，还有一个令传入参数加倍的匿名函数Double。然后我们调用lists:map，通过它在列表的每个元素上调用Double，并从返回结果中得到一个新列表。这是一个绝妙的方法，因为我们会经常用到它，所以Erlang又提供了另一种语法意义相同、但简明得多的方法，也就是列表解析结构。下面就是前面的映射代码使用列表解析的等价形式：

```
4> [Double(X) || X <- Fibs].
[2,2,4,6,10]
```

用大白话描述，这段代码的意义就是："取出列表Fibs中的每个元素X，再把X加倍。"如果你喜欢的话，也可以把中间变量去掉：

```
5> [X * 2 || X <- [1, 1, 2, 3, 5]].
[2,2,4,6,10]
```

表达的意义丝毫没变：取出列表[1,1,2,3,5]中的每个元素X，并计算X*2的值。Erlang的这一特性并不仅仅是语法糖。接下来，我们会写一些更复杂的列表解析。首先，我们用列表解析把map定义得更简洁些：

```
map(F, L) -> [ F(X) || X <- L].
```

其含义是：函数F在列表L上的映射，就是对L中的每个元素X执行F之后，得到的集合F(X)。下面，我们对一个包含了商品名称、数量和价格的商品目录使用列表解析：

```
7> Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}].
[{pencil,4,0.25},{pen,1,1.2},{paper,2,0.2}]
```

假如我要对1美元商品收8美分税，仅用一个列表解析，就能给列表加上税款这一项，并返回一个新列表，像下面这样：

```
8> WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
     {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

先前学过的Erlang概念，在这里仍然管用：这玩意儿还是个模式匹配！这段代码表达的意思是：从列表Cart中取出每一个元组{Product, Quantity, Price}，返回由Product、Price、Quantity以及税款（Price*Quantity*0.08）组成的多元组列表。这段代码在我看来真是妙不可言。现在，只要想改变列表的形式，就可以使用列表解析帮我们自由变化列表。

还有一个例子。比如我现在有个目录，我想对我的高级客户提供内容相同但打了五折的目录。这个目录可能像下面这样，把每类商品从前面那个列表Cart中取出来，但去掉数量：

```
10> Cat = [{Product, Price} || {Product, _, Price} <- Cart].
[{pencil,0.25},{pen,1.2},{paper,0.2}]
```

这就是说，从列表Cart中取出每一个包含Product和Price的元组（忽略第二个属性），返回由Product和Price组成的多元组列表。接下来实现打折：

```
11> DiscountedCat = [{Product, Price / 2} || {Product, Price} <- Cat].
[{pencil,0.125},{pen,0.6},{paper,0.1}]
```

这代码简洁、易懂、威力强大，是一种优美的抽象。

6

实际上，我这里展示的只是列表解析全部威力的冰山一角。它的终极形式比前面提到的这些还要强大得多：

- 列表解析采用的形式是：[表达式 || 子句1, 子句2, …, 子句*N*]；
- 列表解析可拥有任意数量的子句；
- 子句可以是生成器（generator）或筛选器（filter）；
- 筛选器可以是布尔表达式，也可以是返回布尔值的函数；
- 生成器采用`Match <- List`的形式。它在右边列表的各元素上，执行左边式子所表示的模式匹配。

说真的，这些并不算太难。生成器是加进来一些东西，而筛选器是挪出去一些东西。这里，我们能看到Prolog语言带来的大量影响。生成器决定了列表中所有可能出现的值，而筛选器根据给定条件裁剪列表。下面是两个例子：

```
[X || X  <- [1, 2, 3, 4], X < 4, X > 1].
[2,3]
```

用语言来描述，它返回X，X从`[1, 2, 3, 4]`中取值，小于4且大于1。我们还可以使用多个生成器：

```
23> [{X, Y} || X  <- [1, 2, 3, 4], X < 3, Y <- [5, 6]].
[{1,5},{1,6},{2,5},{2,6}]
24>
```

这段代码把X和Y组成元组`{X, Y}`，其中X从`[1, 2, 3, 4]`中取值且小于3，Y从`[5, 6]`中取值。最终满足要求的有两个X值和两个Y值，Erlang计算了它们的笛卡儿积。

这一天的全部内容就是这些。你已经学会了如何用Erlang进行顺序编程。我们现在稍作休息，先复习一下，再做些习题。

6.3.5　第二天我们学到了什么

说老实话，我们今天在Erlang表达式或库等内容上并没学得有多深，但你现在也已经学到了足以写出函数式程序的水平。一开始，你学到的只是一些简单结构，但很快我们就加快了学习的速度。

接下来，我们学习了高阶函数。你可以在列表中使用高阶函数，来对列表进行迭代、筛选和修改。你也学会了如何用`foldl`来把结果汇总起来，就像在Scala中用过的那样。

最后，我们接触了一些高级列表思想。我们在匹配式左边使用`[H|T]`，把列表分解成第一个元素和剩余部分。我们又在匹配式右边使用或单独使用`[H|T]`，从列表头开始构建列表。之后，我们还进一步学习了列表解析。这是一种优雅而强大的抽象，可以通过生成器和筛选器，对列表进行快速变换。

这些语法像是一锅大杂烩。不过由于Erlang有动态类型策略，我们仅用到很少的类型，就能明白这些高级概念。然而，Erlang也有一些东西让人觉得别扭，尤其是不同位置的`case`和`if`子句后面带的分号。

下一节，我们将明白这一节花的这些工夫都是为了什么，我们要解决的是并发问题。

6.3.6 第二天自习

做

- 考虑包含键 – 值元组的列表，如`[{erlang, "a functional language"}, {ruby, "an OO language"}]`。写一个函数，接受列表和键为参数，返回该键对应的值。
- 考虑形如`[{item quantity price}, ...]`的购物列表。写一个列表解析，构建形如`[{item total_price}, ...]`的商品列表，其中`total_price`是`quantity`乘以`price`。
- 加分题：读取一个大小为9的列表或元组，表示井字棋（tic-tac-toe）的棋盘。若胜负已定，则返回胜者（`x`或`o`）；若再没有可走的棋着，则返回`cat`；若两方都还没赢，则返回`no_winner`。

6.4 第三天：红药丸

不少人可能已经知道，在母体（Matrix）中，吃下蓝药丸的人可以浑浑噩噩、但无忧无虑地生活下去；而吃下红药丸的人能看到真相，但有时候，真相是会伤人的。

如今我们所在的整个编程行业，都在大把大把地吞食蓝药丸，就好像来到阿姆斯特丹的牧师之子[①]一样：并发不好用，于是我们就尽量避免用它；我们用了可变状态，因此程序在并发运行的时候就可能导致冲突；我们的函数和方法都有副作用，所以就无法验证其正确性，或是去预测其结果；为了提升性能，我们采用的不是无需共享资源的进程，而是共享了状态的线程，这就要求我们做一些附带工作，以保证每段代码都不出差错。

这几件事堆在一块，后果就是一团乱麻。并发之所以会伤害我们，并不是因为它真有那么难，而是因为我们一直在使用错误的编程模型！

我在这章前面的部分曾提到过，Erlang会把一些简单的事情变难。在一个既没有副作用、又没有可变状态的环境中，你必须将自己的编程习惯全盘改变。也许在许多人眼中，Erlang的这种基于Prolog的语法规则有些匪夷所思，而你却只能默默接受它。然而，此时此刻，收获的季节就在眼前。这颗代表着并发和可靠性的红药丸，现在你尝在嘴里就好似蜜糖。下面，我们就来看看这是为什么。

6.4.1 基本并发原语

Erlang的三种并发原语是：用“`!`”发送消息，用`spawn`产生进程以及用`receive`接收消息。本节，我会教你如何利用这三种原语发送和接收消息，并用它们实现简单的客户端 – 服务器端惯

① 这里所说的牧师之子（preacher's kid），指的是一种美国式的刻板印象：因为牧师是神职人员，家教森严，他们的孩子通常成长于极其严苛的环境之中，因此当牧师的孩子长大成人、独立生活之后，往往会变得离经叛道、放荡不羁，沾染上酗酒甚至吸毒等恶习。至于阿姆斯特丹，我们都知道，那里贩卖、购买毒品都是合法的。文中借“来到阿姆斯特丹的牧师之子”来比喻整个编程行业已对吃蓝药丸司空见惯，也就是对诸多不足之处视而不见，而不是去直面问题并加以改进。

用法。

1. 基本的接收循环

我们先来看一个翻译进程。如果你把一个西班牙语字符串发送给该进程，它会返回字符串的英语译文。一般来说，我们采用的策略是先产生一个进程，再使用一个循环接收并处理消息。

下面是最基本的接收循环：

erlang/translate.erl

```
-module(translate).
-export([loop/0]).

loop() ->
    receive
        "casa" ->
            io:format("house~n"),
            loop();

        "blanca" ->
            io:format("white~n"),
            loop();

        _ ->
            io:format("I don't understand.~n"),
            loop()

end.
```

这个例子比这章之前看过的例子要长一些，所以我们把它分解来看。代码头两行定义了模块`translate`，并输出函数`loop`。之后紧挨着的代码块，就是叫做`loop()`的函数：

```
loop() ->
    ...
end.
```

注意，块中代码调用了三次`loop()`，却什么都没返回，这是有道理的。Erlang对尾递归做了优化，所以只要`loop()`在每个`receive`子句的最后一行，它就不会带来什么开销。但这样做，只是定义了一个永远循环下去的空函数。接下来看看`receive`：

```
receive ->
    ...
```

这个函数接收其他进程发过来的消息。它与Erlang的其他模式匹配结构（如`case`、函数定义等）的运行方式相似。你可以在`receive`后面放上几个模式匹配结构。我们这里的其中一个匹配是这样的：

```
"casa" ->
    io:format("house~n"),
    loop();
```

这是个匹配子句，语法和`case`语句毫无二致。如果收到的消息能匹配字符串`"casa"`，就会执行下面那两行代码。各行代码由`,`分隔开，遇到“`;`”时子句即告终止。上面的代码会显示单词

house，然后再调用loop。（记住，因为loop是最后调用的函数，所以在栈上没有开销。）其他几个匹配子句也都是这样的形式。

现在我们有了一个模块，其中包含receive循环。是时候用它做点什么了。

2. 产生进程

首先，编译这个模块：

```
1> c(translate).
{ok,translate}
```

产生进程要使用spawn函数，它又带有一个函数参数。而这个作为参数的函数，会在一个新的轻量级进程中启动。spawn还会返回一个进程ID（PID）。下面，我们通过translate模块，把参数传入到spawn函数中：

```
2> Pid = spawn(fun translate:loop/0).
<0.38.0>
```

可以看到，Erlang返回的进程ID是<0.38.0>。在命令行界面中，你会看到一对尖括号包含着进程ID。我们这里只涉及产生进程的最基本形式，但你应该知道，还存在其他一些产生形式。你可以用名字注册进程，这样一来，其他进程就可以通过名字找到如公用服务之类的进程，而不是通过进程ID。如果你想拥有运行时更改代码或者说“热插拔”的能力，也可以采用别的spawn形式，如果你要产生一个远程进程，还可以用spawn(Node,function)。不过，这些内容超出了本书的讨论范围。

现在，我们已经用代码块编写了一个模块，还用它产生了一个轻量级进程。我们要做的最后一步，就是把消息传递给它。这也是我们的第三个Erlang原语。

3. 发送消息

使用运算符!，就可以在Erlang中传递分布式消息，就像在Scala那章见过的那样。其用法是Pid ! message，Pid可以是任意一个进程ID，消息也可以是包括原语、列表、元组在内的任意值。现在，我们来发送几条消息：

```
3> Pid ! "casa".
"house"
"casa"
4> Pid ! "blanca".
"white"
"blanca"
5> Pid ! "loco".
"I don't understand."
"loco"
```

这里的每行代码都发送了一条消息。先是receive子句中的io:format打印了一条消息，然后命令行把表达式的返回值（即我们发送的那条消息）也打印了出来。

如果你要做的是把分布式消息发送给命名资源，那你应该用另一种语法：node@server ! message。配置远程服务器并不在本书的讨论范围内，但只要稍加自学，你就能很快明白如何去运行一个分布式服务器。

6

上面的例子展示了这三个基本原语的用法以及如何用它们共同完成简单的异步服务程序。你可能已注意到，这个例子不带有返回值，因此下一节中，我们就要研究如何发送同步消息。

6.4.2　同步消息

有些并发系统是异步运行的，比如电话交谈。发送者只管发送消息，而不必等待响应；有些则是同步运行的，比如上网，我们请求某个网页，在Web服务器把网页发回来之前，我们只能等待响应。下面，我们要把前面那个打印返回值的翻译服务，转变为真正地把翻译过的字符串返回给用户的服务。

为了把消息模型从异步变为同步，我们将采用以下三部分策略。

- 消息服务中的每个receive子句都要匹配一个元组，元组包含请求此次翻译服务的进程ID以及需要翻译的词。有了这ID，我们就知道该把响应发给谁。
- 每个receive子句都要把响应送回给发送者，而不是把结果打印出来。
- 不再使用简单的!原语，而是写一个简单的函数来发送请求和等待响应。

背景资料就是这些。下面，我们看看具体实现的各个方面。

1. 同步接收

第一项任务是给我们的receive子句加几个参数，这意味着要用到元组。有了模式匹配，这件事并不困难。添加参数后的receive子句如下所示：

```
receive
    {Pid, "casa"} ->
        Pid ! "house",
        loop();
        ...
```

它会匹配后面跟着casa一词的元素（该元素应是进程ID），然后它会把house发送给接收者，并回到子句顶端开始下一次循环。

注意这里的模式匹配：元组第一个元素是发送进程的ID，这是receive子句的一种常见形式。除此之外，它和先前的receive子句只有一个较大区别：不是打印结果，而是把结果发回去。然而，和刚才所说的接收消息相比，发送消息的确有点儿难度。

2. 同步发送

说完接收，我们再看看等式的另一边——发送消息。我们先来发一条消息，然后立即进入等待响应状态。既然receiver已提供了进程ID，那么发送一条同步消息就类似下面这样：

```
Receiver !  "message_to_translate",
    receive
        Message -> do_something_with(Message)
    end
```

因为要经常发送消息，所以可以把发送至服务器的请求封装起来，从而简化这一服务。在这个例子中，这个简单的远程过程调用如下所示：

```
translate(To, Word) ->
    To ! {self(), Word},
```

```
    receive
        Translation -> Translation
    end.
```

综合以上这些内容，你可以写一个稍复杂点儿的并发程序：

erlang/translate_service.erl

```
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
        {From, "casa"} ->
            From ! "house",
            loop();

        {From, "blanca"} ->
            From ! "white",
            loop();

        {From, _} ->
            From ! "I don't understand.",
            loop()

end.

translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

其用法如下所示：

```
1> c(translate_service).
{ok,translate_service}
2> Translator = spawn(fun translate_service:loop/0).
<0.38.0>
3> translate_service:translate(Translator, "blanca").
"white"
4> translate_service:translate(Translator, "casa").
"house"
```

我们不过是编译了这段代码、产生循环，再用我们刚刚编写的辅助函数请求同步服务。如你所见，Translator进程如今返回的是经过翻译词语得到的字符串值，而且我们用的是同步消息。

现在，你已经理解了基本的receive循环结构是什么样的。每个进程都有个信箱，receive结构只是把消息从信箱队列中取出来，先用它去匹配某个函数，再去执行这个匹配上的函数。进程利用消息传递机制，在进程间彼此通信。Armstrong博士将Erlang称为真正的面向对象语言，这可不是随便说说。它提供了消息传递和封装等行为，而我们付出的代价，仅仅是失去了可变状态和继承，而继承可以通过高阶函数来模拟。

目前为止，我们只在简单而又理想的条件下运行过Erlang程序，它们并不具备错误恢复能力。尽管Erlang提供了受控异常[①]（checked exception）机制，但接下来，我想带你体验一种不同的错误处理方式。

6.4.3　链接进程以获得可靠性

本节中，我们将学习如何链接进程，以便获得更好的可靠性。Erlang支持把两个进程链接起来。只要其中一个进程终止，它就会将退出信号发送给与之链接的同伴。这样一来，另一个进程会接收到这个信号，并相应地作出反应。

1. 产生链接进程

要想弄明白链接起来的两个进程是如何协作的，我们必须先创建一个易于终止的进程。这里我创建了一个俄罗斯轮盘赌游戏[②]，其中有一把六弹膛枪。只要发送1~6当中的某个数字给枪进程，就会击发数字对应的那个弹膛。若是恰好输入子弹所在的弹膛，则该进程会“杀死”自己。下面是这个游戏的代码：

erlang/roulette.erl

```
-module(roulette).
-export([loop/0]).

% send a number, 1-6
loop() ->
    receive
        3 -> io:format("bang.~n"), exit({roulette,die,at,erlang:time()});
        _ -> io:format("click~n"), loop()
end.
```

这个实现相当简单：我们写了个消息循环，匹配3将执行代码 `io:format("bang~n"), exit({roulette,die,at,erlang:time()};`，这会终止该游戏进程；匹配其他东西则只是打印出一条消息，然后返回循环顶端继续执行。

这样我们就得到了一个简单的客户端–服务器端程序，客户端是命令行，服务器端是`roulette`（轮盘赌）进程，如图6-1所示。

① 受控异常是一种源于Java的异常处理概念。它是相对于运行时异常（runtime exception）而言的，指的是必须用`try...catch`语句在编译时捕获并处理的异常，比如IO异常、SQL异常等。可以说，它们是程序无法直接控制、而依赖于外部条件的一种异常。反之，运行时异常不必在编译时捕获并处理，一般是由于程序员的错误而导致的，比如数组越界异常、空指针异常等。对于这类异常，需要仔细检查错误所在并改正过来。

② 是一种残忍的以生命为赌注的赌博游戏，一般用左轮手枪为道具。假如用的是六弹膛手枪，其中一个弹巢装有子弹，每人轮流向自己击发，则死亡概率均等。该游戏据说最早流行于参加一战的沙俄士兵中，由此得名“俄罗斯轮盘赌”。时至今日，美国等西方国家仍时而有因玩这种游戏丧命的人。

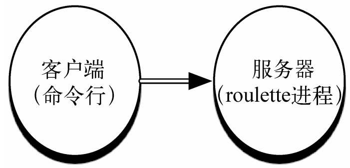


图6-1　简单的客户端–服务器端设计

上面那段代码执行起来会是什么样？请看下面的代码执行过程。

```
1> c(roulette).
{ok,roulette}
2> Gun = spawn(fun roulette:loop/0).
<0.38.0>
3> Gun ! 1.
"click"
1
4> Gun ! 3.
"bang"
3
5> Gun ! 4.
4
6> Gun ! 1.
1
```

这里的问题在于，发送了消息3之后，Gun进程就会终止。因此，3之后发送的消息其实什么都没做。实际上，我们可以判断进程是否终止：

```
7> erlang:is_process_alive(Gun).
false
```

看得出来，这进程真是“没气儿”了，是时候把它搁在板车上拉走了。不过，我们还能把这件事做得更漂亮些。我们可以做个监视器（monitor），告诉我们进程是否终止。我觉得它似乎更像是验尸官（coroner）而不是监视器，因为我们只对进程死没死这件事感兴趣。

下面就是做这件事的代码：

erlang/coroner.erl

```
-module(coroner).
-export([loop/0]).

loop() ->
    process_flag(trap_exit, true),
    receive
        {monitor, Process} ->
            link(Process),
            io:format("Monitoring process.~n"),
            loop();

        {'EXIT', From, Reason} ->
            io:format("The shooter ~p died with reason ~p.", [From, Reason]),
                        io:format("Start another one.~n"),
            loop()
        end.
```

和前面一样，我们写了个receive循环。不过在做其他事之前，这程序必须先注册进程，使之可捕获到退出。不这样做的话，你就无法接收到EXIT消息。

然后，我们要处理接收到的消息。我们可能接收到两种类型的元组：其一是以原子monitor开头的元组，其二是以字符串'EXIT'开头的元组。现在，我们来仔细地看一下这两种类型：

```
{monitor, Process} ->
    link(Process),
    io:format("Monitoring process.~n"),
    loop();
```

这段代码把“验尸官”进程链接到以Process为PID标识的任意进程上。你也可以用spawn_link，产生一个带有链接的进程。现在，如果被监视的进程即将终止，它会发送退出消息给coroner。下面，我们再来看看捕获错误的代码：

```
{'EXIT', From, Reason} ->
    io:format("The shooter died. Start another one.~n"),
    loop()
end.
```

这段代码匹配了退出消息。这消息是个三元组，其中第一个元素是'Exit'，后面跟着From——濒死进程的PID，还有出错原因。我们把濒死进程的PID和出错原因全都打印了出来。下面是完整的运行流程：

```
1> c(roulette).
{ok,roulette}
2> c(coroner).
{ok,coroner}
3> Revolver=spawn(fun roulette:loop/0).
<0.43.0>
4> Coroner=spawn(fun coroner:loop/0).
<0.45.0>
5> Coroner ! {monitor, Revolver}.
Monitoring process.
{monitor,<0.43.0>}
6> Revolver ! 1.
click
1
7> Revolver ! 3.
bang.
3
The shooter <0.43.0> died with reason
{roulette,die,at,{8,48,1}}. Start another one.
```

我们刚才写的这个程序，比起客户端 – 服务器端程序是更进一步了。我们添加了一个监视器进程，如图6-2所示。这样一来，我们就能判断进程何时终止了。

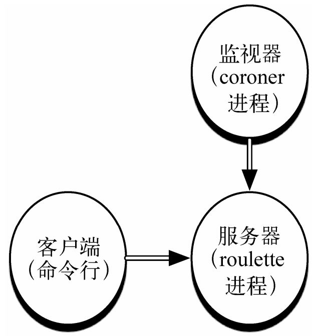


图6-2 添加监视功能

2. 从验尸官到医生

我们还能更上一层楼。如果枪已经注册过（我可没影射现实世界的枪支[①]），游戏玩家就没必要再去获知这支枪的PID了。然后，我们把创建这支枪的代码也挪到了“验尸官”程序中；最后，进程只要一终止，“验尸官”就可以重启该进程。这样一来，我们不必生成很多错误报告，就能获得更好的可靠性。此时此刻，“验尸官”不再是“验尸官”，他变身成了一位“医生”，一位身怀妙手回春之术的“医生”。下面是这个“医生”程序的代码：

erlang/doctor.erl

```
-module(doctor).
-export([loop/0]).

loop() ->
    process_flag(trap_exit, true),
    receive
        new ->
            io:format("Creating and monitoring process.~n"),
            register(revolver, spawn_link(fun roulette:loop/0)),
            loop();

        {'EXIT', From, Reason} ->
            io:format("The shooter ~p died with reason ~p.", [From, Reason]),
                        io:format(" Restarting. ~n"),
            self() ! new,
            loop()
        end.
```

现在，receive语句块匹配了两条消息：一个是new，另一个是和先前相同的EXIT三元组。但与“验尸官”程序相比，两条消息之后的内容都稍有不同。在new代码块中，下面的这行代码堪称绝妙：

① 美国很多州政府都有枪支必须注册的法律，因此这里的“枪已注册”可能会被人误以为有什么一语双关之意。实际上，这里指的并不是枪支注册，而是枪进程的注册。不过作者举这样一个例子，其实也是想开个小玩笑。

```
register(revolver, spawn_link(fun roulette:loop/0)),
```

从里面那个括号开始看，我们用了spawn_link来产生进程。这种方式不仅能产生进程，还能把进程链接起来。这样当roulette进程终止时，doctor进程就能获得通知。再看外面的括号，我们注册了那个新产生进程的PID，把它和revolver原子关联起来。现在，用户可以用revolver ! message这种形式发送消息给新产生的进程。从今往后，我们再也不需要PID了。同样，EXIT的匹配代码块也变得更加智能了。下面是这个块里新加的一行代码：

```
self() ! new,
```

这里我们把消息发给自己，新产生并注册了一把枪。这样一来，这游戏玩起来就简单多了：

```
2> c(doctor).
{ok,doctor}
3> Doc = spawn(fun doctor:loop/0).
<0.43.0>
4> revolver ! 1.
** exception error: bad argument
     in operator  !/2
        called as revolver ! 1
```

显然，我们还没有创建进程，因此出现了一个错误。现在我们创建并注册进程：

```
5> Doc ! new.
Creating and monitoring process.
new
6> revolver ! 1.
click
1
7> revolver ! 3.
bang.
3
The shooter <0.47.0> died with reason {roulette,die,at,{8,53,40}}.
    Restarting.
8> Creating and monitoring process.
8> revolver ! 4.
click
4
```

这里我们是在Doctor里创建的revolver，这其实和“医生”的身份不太搭。还有一点和前面不同，我们击发手枪，用的是revolver原子而不是枪的PID来发送消息。此外，可以看一下行号为8的那行代码，我们实际上创建并注册了一把新的左轮手枪。这类程序的整体结构通常如图6-2所示，“医生”在这里扮演了比“验尸官”更为主动的角色。

对Erlang，我们不过蜻蜓点水一般浅尝辄止。但我希望你明白，Erlang可轻松创建比过去健壮得多的并发系统。而且你根本看不见多少错误处理。只要某个东西一崩溃，它就会启动一个新的。写个监视其他进程的监视器也没什么难度。实际上，Erlang的基础库拥有足够的工具。这些工具既可以用来编写监视器服务，也可以是出现致命错误时自动重启的“持久”进程。

6.4.4 第三天我们学到了什么

第三天开始，你对Erlang能做什么这一点有了更深理解。我们一上来就学到了三种并发原语：send、receive、spawn。我们通过编写异步版本的翻译程序，展示了基本的消息传递机制是如何工作的。然后，我们编写了一个简单的辅助函数，把发送和接收封装在一起，用这个封装后的发送和接收来模拟远程过程调用。

接下来，我们通过把进程链接起来的方式，展示一个进程终止时如何通知另一进程。我们还学会了用另一进程来监控某个进程，以获得更好的可靠性。虽然我们采用的编程思想也可用在容错系统的构建当中，但我们这里的系统并不是容错系统。Erlang分布式通信的运行机制很像进程间通信。我们可以链接两台计算机上的两个进程，这样备份机就可以监视主机，并在主机出问题时接管任务。

现在，我们在自习中检验一下学到的内容。

6.4.5 第三天自习

下面的习题比较简单，我会补充一些加分题，给你的自习加一些挑战。

开放电信平台（Open Telecom Platform，OTP）是个强大的软件包，你可以用它所提供的很多工具来构建一个分布式的并发服务。

找

- 可以在进程终止时重启它的OTP服务。
- 构建简单的OTP服务器的文档。

做

- 监视`translate_service`，并在它终止时重启它。
- 如果`Doctor`进程终止，使其重启自身。
- 写一个监视`Doctor`监视器的监视器。如果其中某个监视器终止，则重启它。

下面的加分题需要你稍做研究方可完成：

- 创建一个基本的OTP服务器，可以把消息记录到文件中。
- 让`translate_service`跨网络运行。

6.5 趁热打铁

本章一开始我就曾说过，Erlang既可将难事化易，也可将易事化难。它的Prolog风格语法，让那些习惯了一系列C风格语言的程序员们感到不太自在。同时，它的函数式编程范型也带来了一大堆难题。

然而，随着硬件设计不断更新发展，并发编程日趋重要，Erlang的一些核心功能也会变得举足轻重起来。它的某些功能颇具哲理。轻量级进程的设计思想与Java的线程和进程模型设计是背道而驰的。“就让它崩溃”的哲学不仅极大简化了代码，而且要求在虚拟机层提供基本支持，这

6

在其他系统中是不存在的。下面，我们来分别讨论Erlang的核心优势和不足之处。

6.5.1　核心优势

发自内心地说，Erlang就是并发和容错的代名词。因为处理器的设计者想到了分布式多处理器方法，所以最新的编程技术也需要提高。Erlang的威力主要针对这一代程序员将要面对的最重要的领域。

1. 动态和可靠性

首先，Erlang是为可靠性而生的。Erlang的核心库久经考验，而用它写的应用程序在当今世界上也属于最可靠和最好用的一类。令人印象最为深刻的是，Erlang的设计者并没有为了获得这种可靠性而牺牲掉使Erlang如此高效的动态类型策略。Erlang依靠的不是编译器所提供的人工的“安全网”，而是链接并发进程的能力，这样既可靠又简单。我十分惊叹于Erlang没有用到操作系统的那些蹩脚技术，就可以轻易地构建可靠的监视器。

你在Erlang中发现的那些为保证可靠性而做的妥协，在我看来也非常激动人心和独一无二。拿Java语言和虚拟机来说，它没有提供一组正确的原语，因此无法达到Erlang的那种性能和可靠性。同时，BEAM上的库也体现了Erlang的可靠性哲学，所以用它们构建可靠的分布式系统也就容易了很多。

2. 轻量级、无共享资源的进程

Erlang的另一处闪光点是其底层的进程模型。Erlang的进程是轻量级的，因此Erlang程序员会经常使用它们。Erlang是建立在强制要求不变性的哲学之上的，因此Erlang程序员构建的系统从根本上就不太可能因为互相冲突而产生致命错误。Erlang拥有消息传递范型和原语，因此它可以轻易地写出带有一定的分离性的应用程序，而这在其他面向对象应用中是较为罕见的。

3. OTP——企业级的库

因为Erlang是在电信企业中成长起来的语言，对可用性和可靠性都有很高要求，所以可以说，它在这个领域已有20年的开发经验。这个领域中，最主要的库是开放电信平台（Open Telecom Platform，OTP）。你可以找到帮你构建各种应用的库，包括处在监视状态下的持久进程、访问数据库的连接、分布式应用等。OTP有整套的Web服务器以及众多工具都可用于构建电信应用。

这些库特别出色的一点在于：容错性、可扩展性、事务完整性、热插拔性，全都是内置特性。你不必再为它们操心。你完全可以利用这些特性构建自己的服务器进程。

4. 就让它崩溃

在Erlang中，处理并行进程不会有副作用，这是“就让它崩溃”策略带来的结果——你不必理会进程为什么崩溃，因为你只要重启它就好。这是一种函数式编程模型，Erlang的分布式策略也由此得到了增强。

和本书其他语言一样，Erlang也并非毫无瑕疵，问题无所不在，只是问题的种类有所改变而已。面对这些问题，就连特工Smith都无法做到尽善尽美。

6.5.2 不足之处

Erlang的根本问题是用户群太小，这源于它根深蒂固的小众语言定位。在大多数程序员看来，它的语法很难称得上平易近人。此外，Erlang的函数式语言范型也让它有些格格不入，这也同样阻碍了它获得广泛应用；最后，迄今为止，最好的Erlang实现都在BEAM上，而不是Java虚拟机上。下面我们就来进一步阐述一下这几点。

1. 语法

和电影一样，语法也是个见仁见智的东西。除此之外，Erlang还有一些问题，就是那些不刻意挑刺儿的人也能看到的。下面，我们就来看看其中的两个问题。

有一件事很有趣：Erlang最核心的优势和劣势，都源于它以Prolog为基础。对于大多数编程者来说，Prolog是晦涩难懂的，其语法颇有些别扭的味道。如果从其他语言过渡到Prolog，一点点语法糖无疑对降低学习难度大有帮助。

本章，我提到了`if`和`case`语法结构的问题。它们的语法规则是合乎逻辑的——在语句之间用分隔符分开。然而，如果不改变标点，你就无法改变`case`、`if`或`receive`块的顺序，所以说这规则又不太实用。这些语法上的限制是没有必要的。Erlang还有一些别的古怪之处，比如说，符合条件的数字数组会显示为字符串。若除掉这些问题，Erlang必将大有长进。

2. 整合

刚才说过，Prolog的传承者这一身份，既会带给Erlang优点，也会带来缺点。同样，不在JVM上实现Erlang也是一把双刃剑。最近，一个基于JVM的虚拟机Erjang取得了一定进展，但仍未达到JVM上的最佳选择的水准。的确，JVM是有些拖泥带水，比如无法满足Erlang所要求的进程和线程模型等，但在JVM上实现也有诸多好处，包括各种各样的Java库以及数以十万计的可用部署服务器等宝贵财富。

6.5.3 最后思考

编程语言的成功是无法预测的。Erlang在市场推广上面临着严峻挑战，吸引Java程序员过来使用Lisp风格的编程范型和Prolog风格的语法绝非易事。看样子，Erlang正在慢慢地积蓄力量，因为它能在正确的时间和正确的地点解决正确的问题。在这场Neo与特工Smith之间的战斗中，我认为特工Smith成功或失败的几率各占一半。

第 7 章 Clojure

做或不做，不要尝试。

——Yoda（尤达大师）

Clojure是JVM上的Lisp实现。Lisp复杂强大，是计算机领域里最早和最新的编程语言之一。许多Lisp方言都曾尝试挤进主流语言的行列，却都无功而返。即便是对今天的开发者而言，其语法和编程模型也有些难以消化。即便如此，Lisp的特质仍叫人禁不住去重温，去回味，新的方言层出不穷，一些编程领域最好的院校也用Lisp语言来帮助学生们塑造创新、开放的思维方式。

从很多方面来看，Clojure就像是睿智的功夫大师，神隐山脉的先知或是高深莫测的绝地师父。想想Yoda。在《星球大战系列之五：帝国反击战》[①]中，Yoda是一位小巧、可爱的配角。他总是使用“倒装”语序说话，但却意味高深[②]，就像Lisp所使用的前缀表示法（相信过一会儿你就会明白）。他小巧到难以辨别，就像Lisp的语法不过是一些括号和符号。但是和Yoda一样，它绝非看上去那么简单。Yoda和Lisp年岁都很高，拥有的智慧（例如开头的引语）经过时间磨砺与烈火考验。Lisp宏和高阶编程单元如同Yoda掌握的内在原力，看似无人能掌控。从许多角度讲， Lisp开创了一切。在深入Clojure之前，让我们先来谈谈Lisp，然后再来了解Clojure的激动人心之处。

7.1 Clojure 入门

说白了，Clojure就是一种Lisp方言，受Lisp语言限制，但也拥有Lisp强大的力量，了解Clojure要从了解Lisp开始。

7.1.1 一切皆Lisp

Lisp是继Fortran之后最古老的、商业上依然活跃的编程语言。它是函数式语言，但不是纯函

① *Star Wars Episode V:The Empire Strikes Back*。导演：George Lucas（1980年）。发行商加利福尼亚州比佛利山20世纪福斯影片公司（2004年）（译者注：这部电影中译名为《星球大战 V：帝国反击战》，是一部科幻电影。片中的Yoda是德高望重的“绝地”议会长老。九百岁的Yoda授徒长达8个多世纪，是一位具有强大“原力”与高洁品德的大师。）

② Yoda的母语可能是一种具有“宾、主、谓”结构的语言，所以他常常会这样说：“Take you to him I will.”（带你去见他，我会。）

数式语言。Lisp是LISt Processing的缩写，很快你就会明白这个名字的由来。Lisp拥有一些非常有趣的特性。

- Lisp是一种列表语言。函数调用时，取列表第一个元素作函数，列表其余元素作参数；
- Lisp使用自有数据结构表示代码。一些Lisp追随者称之为*数据即代码*（data as code）。

结合这两种特性，所得到是一种特别适合元编程的语言。可以把代码组织成像类里的方法一样。再用这些对象构成树，就能得到一个基本的对象模型。也可以构造一个基于原型的代码组织结构，其中为数据和行为预留下扩展槽。还可以构造出一个纯函数式实现。正是这种灵活性让Lisp几乎能支持任何编程范型。

在《黑客与画家》一书中，Paul Graham记载了一个小团队用Lisp及其强大的编程模型打败大公司的故事。他们相信Lisp提供了极为显著的编程优势。实际上，与大公司相比，他们更关注在招聘启事中对Lisp和其他更高级编程语言提出要求的那些初创公司。①

目前最主要的Lisp方言是Common Lisp和Scheme。Scheme和Clojure都来自lisp-1方言，而Common Lisp来自lisp-2方言。这两大家族之间主要的区别在于命名空间的工作方式。Common Lisp用不同的命名空间区分函数和变量，Scheme则不区分。讲完了Lisp这边，再来讲讲Java这边。

7.1.2 JVM

每一种Lisp方言都有其迎合的群体。对Clojure而言，最重要的特征之一就是JVM。看看Scala就能明白，拥有市场上非常成功的部署平台可以带来完全不同的效果。你不必为了使用Clojure而向运维人员推销Clojure服务器。尽管语言本身相对较新，但却可以利用成百上千的Java类库来满足你的任何需要。

在本章中，你将会看到各种证明JVM存在的证据，在调用中、在类库中、在我们创建的构件中。另外你还有机会看到其灵活自由的一面。Clojure是函数式的，因此可以在代码中应用更高级的理念。Clojure是动态类型的，因此代码可以写得更简练，更易于阅读，编写时乐趣无穷。除此以外，Clojure还拥有Lisp非凡的表现力。

Clojure和Java谁也离不开谁。Lisp需要Java虚拟机所能提供的市场份额，而Java社区也需要注入新的活力。

7.1.3 为并发更新

构成Clojure方程式的最后一块就是类库集合。Clojure是一门函数式语言，强调函数无副作用。但当你一定要使用可变状态时，语言支持的大量概念也会提供帮助。事务内存（transactional memory）工作起来类似事务型数据库，并提供安全、并发的内存访问。代理（agent）支持对可变资源访问的封装。我们会在第三天解释这些内容。

你是不是已经等不及了？让我们赶快开始吧。

① 《黑客与画家》中文版第179页。该书由人民邮电出版社出版。

7.2　第一天：训练 Luke

在《星球大战》中，学徒Luke（卢克）跟随Yoda按绝地武士的方式进行高阶训练。他已经开始在另一位大师的指引下投入训练。像Luke一样，其实你也早已开始函数式语言的练习。你使用过Ruby的闭包，并完成了Scala和Erlang的高阶函数学业。本章中，你将学习如何在Clojure里应用这些概念。

请访问Clojure网站，网址为http://www.assembla.com/wiki/show/clojure/Getting_Started。按照上面的指示在你的平台上安装Clojure并准备好你喜欢的开发环境。我使用的是Clojure 1.2的prerelease版本，等你拿到此书时这个版本应该已经是稳定版了。[①]你可能需要首先安装Java平台，但如今的多数操作系统都已预装了Java。我使用leiningen[②]工具来管理Clojure项目和Java配置。它能帮助我方便地构建项目，而且还不用关心像`classpath`这样与Java相关的细节。leiningen安装好后，就可以创建新项目了：

```
batate$ lein new seven-languages
Created new project in: seven-languages
batate$ cd seven-languages/
seven-languages batate$
```

接着，可以启动Clojure的控制台，也称之为repl：

```
seven-languages batate$ lein repl
Copying 2 files to /Users/batate/lein/seven-languages/lib
user=>
```

……然后就可以开始编程了。在后台，leiningen自动完成了依赖项的安装，然后调用Java并传入Clojure的jar包和一些参数。你也许需要通过其他方式来启动控制台。从现在开始，我只会告诉你启动repl（read-eval-print loop，读取–求值–打印循环）。

完成了上面的工作后，你会看到一个初始的控制台命令行窗口。当我要求你执行代码时，可以使用这个控制台，或者也可以使用任何支持Clojure的IDE和编辑器。

下面来写点儿代码：

```
user=> (println "Give me some Clojure!")
Give me some Clojure!
nil
```

嗯，这说明控制台工作正常。在Clojure中，任何函数调用都需要用括号将其包起来。括号里的第一个元素是函数名，剩下的是参数。你也可以嵌套调用函数。让我们举一些数学计算的例子来演示如何调用函数。

7.2.1　调用基本函数

```
user=> (- 1)
```

① 翻译本章时，Clojure版本号已经升级到1.3。
② http://github.com/technomancy/leiningen。——原书注

```
-1
user=> (+ 1 1)
2
user=> (* 10 10)
100
```

这些只是最基本的数学计算。除法会更有趣一点儿：

```
user=> (/ 1 3)
1/3
user=> (/ 2 4)
1/2
user=> (/ 2.0 4)
0.5
user=> (class (/ 1 3))
clojure.lang.Ratio
```

Clojure中有一个基本数据类型叫做`ratio`。这是一个很棒的特性，因为它支持延迟计算以避免精度损失。如果有需要，也可以用浮点数进行运算。可以像下面这样计算余数：

```
user=> (mod 5 4)
1
```

`mod`是取模操作符的简称。这种表示法称为*前缀表示法*。之前学过的各种编程语言所支持的是中缀表示法，即操作符放在两个操作数的中间，例如4 + 1 – 2。许多人更喜欢中缀表示法，因为它符合我们的使用习惯。我们早已习惯于使用这种方式来表示数学算式。经过前面的热身，你应该已经逐渐适应用前缀表示法书写的代码行了。尽管用这种方式做数学运算略显笨拙，但它是可行的。带括号的前缀表示法确有它的优势。考量下面的表达式：

```
user=> (/ (/ 12 2) (/ 6 2))
2
```

这里不存在歧义。Clojure会按照括号的顺序来执行语句。再来看看下面这个表达式：

```
user=> (+ 2 2 2 2)
8
```

你也可以往算式上添加元素。甚至在做减法和除法时也能写成这种风格：

```
user=> (- 8 1 2)
5
user=> (/ 8 2 2)
2
```

用传统的中缀法表示上面的表达式，要写成`(8 - 1) - 2`和`(8 / 2) / 2`。Clojure中每次只使用两个操作数的等价形式为：`(- (- 8 1) 2)`和`(/ (/ 8 2) 2)`。使用这种简单的操作符求值还能得到一些令人惊讶的强大效果：

```
user=> (< 1 2 3)
true
user=> (< 1 3 2 4)
false
```

这太棒了。对任意数量的参数所组成的列表，只要用一个简单的操作符就能知道它是否已排序。

除了前缀表示法和新颖的多参数列表（multiple parameter lists）外，Clojure的语法其实特别

7

简单。现在让我们再看看类型系统，探究一下强类型和类型转换：

```
user=> (+ 3.0 5)
8.0
user=> (+ 3 5.0)
8.0
```

Clojure替我们完成了类型转换的工作。一般地，Clojure支持强类型和动态类型。让我们再进一步，看看Clojure中的一些基本构成部分，即形式[①]（form）。

可以把形式看作是语法的一部分。当Clojure解析代码时，它会将代码分解成很多份称为形式的片段。然后Clojure会编译或者解释执行代码。不用区分代码和数据，因为在Lisp中这二者是同一种东西，是等价的。在本章中，你将会见到布尔值、字符、字符串、集合（set）、映射表（map），以及向量（vector），这些都是形式。

7.2.2　字符串和字符

前面简单介绍过字符串，这一小节我们再来深入了解一下。Clojure中字符串由双引号括起来，并使用C语言风格的转义字符，和Ruby一样：

```
user=> (println "master yoda\nluke skywalker\ndarth vader")
master yoda
luke skywalker
darth vader
nil
```

这没有什么特殊之处。另外提一下，至今为止，我们使用`println`时都只传了一个参数，其实这个函数也适用于传入零或多个参数的情况，如果没有指定参数则打印一个空行，传递多个参数则将多个值连起来打印。

在Clojure中，可以用`str`函数将其他类型的数据转换成字符串：

```
user=> (str 1)
"1"
```

如果目标是Java类型，`str`会调用底层的`toString`方法。该函数可接受多个参数，如下所示：

```
user=> (str "yoda, " "luke,  " "darth")
"yoda, luke,  darth"
```

这样Clojure开发者可以用`str`函数来连接字符串。特别方便的是，它还能拼接非字符串项：

```
user=> (str "one: " 1 " two: " 2)
"one: 1 two: 2"
```

要在双引号之外表示一个字符需用字符\，像这样：

```
user=> \a
\a
```

也能用`str`函数拼接字符串：

```
user=> (str \f \o \r \c \e)
"force"
```

① 参考《实用Common Lisp编程》一书中的Lisp形式。该书由人民邮电出版社出版。

我们来作个比较：

```
user=> (= "a" \a)
false
```

可以看出字符并不是长度为1的字符串。

```
user=> (= (str \a) "a")
true
```

不过可以将一组字符转换成字符串。有关字符串的讲解已经足够了。下面我们来说说布尔表达式。

7.2.3 布尔值和表达式

Clojure支持强类型和动态类型。动态类型意味着类型在运行时求值。你已经见过一些类型了，现在我们再来探讨一下这个话题。布尔值是表达式求值的结果：

```
user=> (= 1 1.0)
true
user=> (= 1 2)
false
user=> (< 1 2)
true
```

和本书中提到的其他编程语言一样，`true`是一个符号，但它也是别的东西。Clojure中的类型与底层Java类型系统是统一的，可以用`class`函数获得其底层类，这是布尔值的类：

```
user=> (class true)
java.lang.Boolean
user=> (class (= 1 1))
java.lang.Boolean
```

这里可以瞥见JVM的踪迹。随着学习的逐渐深入，这种类型策略的便捷性会逐渐显现。可以在很多表达式中使用布尔值的结果，下面是一个简单的`if`语句：

```
user=> (if true (println "True it is."))
True it is.
nil
user=> (if (> 1 2) (println "True it is."))
nil
```

7

与Io类似，我们把代码作为第二个参数传递给`if`。特别方便的是，Lisp允许把数据当作代码。这样我们可以把代码分成多行，让它看起来更漂亮些：

```
user=> (if (< 1 2)
  (println "False it is not."))
False it is not.
nil
```

还能添加一个相当于`else`分支的部分作为第三个参数：

```
user=> (if false (println "true") (println "false"))
false
nil
```

现在，让我们看看还有什么能用作布尔值。首先，在Clojure中什么会取值nil呢？

```
user=> (first ())
nil
```

啊哈，非常简单。这个符号就叫nil。

```
user=> (if 0 (println "true"))
true
nil
user=> (if nil (println "true"))
nil
user=> (if "" (println "true"))
true
nil
```

0和""都是true，但nil不是。另外一些布尔表达式会在需要时再予以介绍。接下来，让我们看看更复杂的数据结构。

7.2.4　列表、映射表、集合以及向量

在所有函数式语言中，诸如列表（list）、元组（tuple）这样的核心数据结构往往承担着更繁重的工作。Clojure中，类似的三员大将分别是列表、映射表（map）和向量（vector）。我们先从你最熟悉的集合类型开始，那就是列表。

1. 列表

列表是元素的有序集合。这些元素可以是任何东西，不过在Clojure中，习惯将列表用作代码，把向量用作数据。为了避免产生困惑，我会简单浏览一下列表作为数据的例子。由于列表是被当作函数进行求值的，因此不能这么写：

```
user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer
    cannot be cast to clojure.lang.IFn (NO_SOURCE_FILE:0)
```

如果确实想要创建一个由1、2和3组成的列表，下面两种方式任选其一：

```
user=> (list 1 2 3)
(1 2 3)
user=> '(1 2 3)
(1 2 3)
```

接着就可以照例操作列表了。上面第二个形式叫做quoting（引用）。列表最主要的操作有四个，分别是first（头元素）、rest（除头部以外的列表）、last（最后一个元素）以及cons（给定一个头元素和尾部，创建一个新列表）：

```
user=> (first '(:r2d2 :c3po))
:r2d2
user=> (last '(:r2d2 :c3po))
:c3po
user=> (rest '(:r2d2 :c3po))
(:c3po)
```

```
user=> (cons :battle-droid '(:r2d2 :c3po))
(:battle-droid :r2d2 :c3po)
```

当然可以和高阶函数组合使用它们，不过在遇到序列（sequence）时才会用到这种方式。现在，让我们来看看列表的“兄弟”向量。

2. 向量

和列表一样，向量也是元素的有序集合。向量是为随机访问优化的。用方括号括起来，像这样：

```
user=> [:hutt :wookie :ewok]
[:hutt :wookie :ewok]
```

列表作代码，向量作数据。可以像下面这样获取各种元素：

```
user=> (first [:hutt :wookie :ewok])
:hutt
user=> (nth [:hutt :wookie :ewok] 2)
:ewok
user=> (nth [:hutt :wookie :ewok] 0)
:hutt
user=> (last [:hutt :wookie :ewok])
:ewok
user=> ([:hutt :wookie :ewok] 2)
:ewok
```

注意，向量也是函数，取下标为参数。可以合并两个向量，像这样：

```
user=> (concat [:darth-vader] [:darth-maul])
(:darth-vader :darth-maul)
```

你可能注意到repl打印出来的不是向量而是列表。许多返回集合的函数都使用了称为序列的Clojure抽象。我们会在第二天里进一步学习它们。现在，只要明白Clojure返回的是序列并且在repl中以列表的形式展现就可以了。

当然Clojure也允许从向量中获取头元素和尾部：

```
user=> (first [:hutt :wookie :ewok])
:hutt
user=> (rest [:hutt :wookie :ewok])
(:wookie :ewok)
```

在进行模式匹配时这些特性都用得上。列表和向量都是有序集合。接下来让我们看看无序集合，set和映射表。

7

3. Set[①]

set（集合）是元素的无序集合。其实set有一个固定的顺序，不过与实现方式相关，因此不能依赖这个顺序。set用#{}包起来，像这样：

```
user=> #{:x-wing :y-wing :tie-fighter}
#{:x-wing :y-wing :tie-fighter}
```

可以把它们赋给变量名spacecraft再来操作它们：

① 由于set的常见翻译“集合”与collection冲突，为了避免混淆，在这一节里，我们直接用set而不做翻译。除此外的译文中，如不做特殊说明，集合一词都是指collection。

```
user=> (def spacecraft #{:x-wing :y-wing :tie-fighter})
#'user/spacecraft
user=> spacecraft
#{:x-wing :y-wing :tie-fighter}
user=> (count spacecraft)
3
user=> (sort spacecraft)
(:tie-fighter :x-wing :y-wing)
```

也可以创建一个排序set，它以任意次序接受元素，并按照一个固定的次序返回元素：

```
user=> (sorted-set 2 3 1)
#{1 2 3}
```

可以合并两个set，像这样：

```
user=> (clojure.set/union #{:skywalker} #{:vader})
#{:skywalker :vader}
```

求差集：

```
(clojure.set/difference #{1 2 3} #{2})
```

在继续下一部分之前，还有最后一个超便利的小妙招透露给你。其实set也是函数。比如`#{:jar-jar, :chewbacca}`，它不仅是一个set，同时还是一个函数，可用来判别元素是否属于该set的成员，例如：

```
user=> (#{:jar-jar :chewbacca} :chewbacca)
:chewbacca
user=> (#{:jar-jar :chewbacca} :luke)
nil
```

当把set当作函数使用时，如果参数属于set则返回参数。到此已经介绍完了有关set的基本知识。接下来看看映射表。

4. 映射表

映射表就是键值对。在Clojure中，用大括号来表示映射表，像这样：

```
user=> {:chewie :wookie :lea :human}
{:chewie :wookie, :lea :human}
```

这就是映射表（即键值对），它的问题很难看清楚，即使写了奇数个键值也不太容易发现，从而导致错误：

```
user=> {:jabba :hut :han}
java.lang.ArrayIndexOutOfBoundsException: 3
```

Clojure通过使用逗号作为空白符来解决这个问题，像这样：

```
user=> {:darth-vader "obi wan", :luke "yoda"}
{:darth-vader "obi wan", :luke "yoda"}
```

以冒号:开始的词是关键词，类似Ruby中的符号或者Prolog或Erlang中的原子。类似命名事物的形式在Clojure里有两种：关键词和符号。符号指向其他东西，而关键词指向自身。`true`和`map`就是符号。可以参照在Erlang中使用原子的方法，使用关键词去命名这类实体，例如映射表中的属性等。

我们定义一个叫mentors的映射表：

```
user=> (def mentors {:darth-vader "obi wan", :luke "yoda"})
#'user/mentors
user=> mentors
{:darth-vader "obi wan", :luke "yoda"}
```

可以用键获取对应的值：

```
user=> (mentors :luke)
"yoda"
```

映射表本身也是函数。关键词同样也是函数。

```
user=> (:luke mentors)
"yoda"
```

函数:luke的作用是在映射表中查找自身。虽然看起来有点儿怪，但却很实用。和Ruby一样，在Clojure中可以使用任何数据类型作为键或者值。可以用merge函数合并两个映射表：

```
user=> (merge {:y-wing 2, :x-wing 4} {:tie-fighter 2})
{:tie-fighter 2, :y-wing 2, :x-wing 4}
```

对于一个散列值同时出现在两个映射表中的情况，还可以声明一个操作符：

```
user=> (merge-with + {:y-wing 2, :x-wing 4} {:tie-fighter 2 :x-wing 3})
{:tie-fighter 2, :y-wing 2, :x-wing 7}
```

本例中，使用操作符+合并了与x-wing相关联的值4和3。给定一个关联，还可以通过加入新键值对来创建新的关联，像这样：

```
user=>(assoc {:one 1} :two 2)
{:two 2, :one 1}
```

还可以创建排序映射表。排序映射表以任意次序接收条目，然后再输出排好序的结果，如下所示：

```
user=> (sorted-map 1 :one, 3 :three, 2 :two)
{1 :one, 2 :two, 3 :three}
```

以后我们还会在数据中逐渐增加更多的结构。现在，让我们把目光转移到用于增加行为的形式上来，聊聊函数。

7

7.2.5 定义函数

函数是所有Lisp语句的核心。用defn定义函数。

```
user=> (defn force-it [] (str "Use the force," "Luke."))
#'user/force-it
```

定义函数的最简形式是(defn [params] body)。我们定义了一个没有参数的函数force-it。这个函数只是把两个字符串拼接起来。调用方式和其他任何函数一样：

```
user=> (force-it)
"Use the force,Luke."
```

如果愿意的话，还可以加上描述函数的字符串：

```
user=> (defn force-it
             "The first function a young Jedi needs"
             []
             (str "Use the force," "Luke"))
```

然后可以用doc函数调出该文档：

```
user=> (doc force-it)
-------------------------
user/force-it
([])
  The first function a young Jedi needs
nil
```

接下来添加参数：

```
user=> (defn force-it
             "The first function a young Jedi needs"
             [jedi]
             (str "Use the force," jedi))
#'user/force-it
user=> (force-it "Luke")
"Use the force,Luke"
```

顺便提一下，对任何提供了此描述文档的函数都可以使用doc功能：

```
user=> (doc str)
-------------------------
clojure.core/str
([] [x] [x & ys])
  With no args, returns the empty string. With one arg x, returns
  x.toString().  (str nil) returns the empty string. With more than
  one arg, returns the concatenation of the str values of the args.
nil
```

我们已经定义了一个基本函数，接下来看看参数列表。

7.2.6　绑定

和我们所见过的其他编程语言一样，绑定是指按照实参对形参进行赋值的过程。Clojure特点是它能访问到实参中的任意部分，并能把这部分作为形参，这个功能非常有用。举个例子，比如一条线段，使用由点组成的向量表示，像下面这样：

```
user=> (def line [[0 0] [10 20]])
#'user/line
user=> line
[[0 0] [10 20]]
```

可以构造一个函数来访问线段的结尾，像下面这样：

```
user=> (defn line-end [ln] (last  ln))
#'user/line-end
user=> (line-end line)
[10 20]
```

实际上可能并不需要整条线段。如果形参能绑定到线段的第二个元素上，那就更好了。在Clojure中这很容易实现：

```
(defn line-end [[_ second]] second)
#'user/line-end
user=> (line-end line)
[10 20]
```

这里用到一个概念叫解构（destructuring）。即将一个数据结构拆解开来并从中提取出需要的部分。下面我们来详细探讨绑定。对于`[[_ second]]`，外层方括号用来定义形参向量，内层方括号指明将要绑定的是列表的元素还是向量的元素。`_`和`second`分别表示单独的元素，通常用`_`（下划线）表示被忽略的参数。也就是说，这个函数的形参分别是：`_`对应第一个实参中的第一个元素，`second`对应第一个实参中的第二个元素。

绑定可以嵌套。比方说有个井字棋（又叫一字棋， tic-tac-toe）棋盘。现在我们想返回棋盘正中间那个方框里的值。假设棋盘表示为每行三子共三行，像下面这样：

```
user=> (def board [[:x :o :x] [:o :x :o] [:o :x :o]])
#'user/board
```

这样一来，可以取出第二行的第二个元素，如下：

```
user=> (defn center [[_ [_ c _] _]] c)
#'user/center
```

多漂亮！本质上就是嵌套使用同一个概念。让我们分解开来看看。对于 `[[_ [_ c _] _]]` 绑定，我们只绑定了一个形参到实参上，即 `[_ [_ c _] _]`。这个形参的含义是，忽略第一个和第三个元素，二者分别对应棋盘顶部和底部两行。取中间那行，即`[_ c _]`。我们期望看到另一个列表，然后再提取出列表正中间的一个元素：

```
user=> (center board)
:x
```

有两种方法来简化这个函数。首先，不需要列出目标实参之后的任何通配符实参：

```
(defn center [[_ [_ c]]] c)
```

其次，解构除了可以发生在参数列表中，还可以发生在`let`语句中。对任何Lisp，使用`let`语句都能将一个变量绑定到一个值上。因而可以利用`let`函数对`center`函数的调用方隐藏解构：

```
(defn center [board]
 (let [[_ [_ c]] board] c))
```

`let`函数需要两个参数。第一个参数是一个向量，包含需要进行绑定的符号（`[[_ [_c]]]`），跟上想要绑定的值（`board`）。第二个参数可以是将使用值（`c`）的表达式（我们只是返回`c`）。两种形式都能产生等价的结果。采用哪种形式完全取决于要在哪里实现解构。我会向你展示一系列使用`let`函数的简短样例，而这些例子同样可以在参数列表中使用。

解构映射表：

```
user=> (def person {:name "Jabba" :profession "Gangster"})
#'user/person
user=> (let [{name :name} person] (str "The person's name is " name))
"The person's name is Jabba"
```

7

同时处理映射表和向量：

```
user=> (def villains [{:name "Godzilla" :size "big"} {:name "Ebola" :size "small"}])
#'user/villains
user=> (let [[_ {name :name}] villains] (str "Name of the second villain: " name))
"Name of the second villain: Ebola"
```

上面例子中我们绑定了一个向量，跳过第一个元素并提取出第二个映射表中的name值。你可以从这看出Lisp对Prolog，以及（推而广之）Erlang的影响。解构基本上就是模式匹配的一种形式。

7.2.7　匿名函数

在Lisp中，函数就是数据。由于代码和其他任何类型数据一样都是数据，因此高阶函数从语言创建之初就被加入到其中。匿名函数可以创建没有名字的函数。这是本书中提到的每一种语言都具备的基本能力。在Clojure中，用函数fn来定义高阶函数。通常会忽略函数名，所以调用形式看起来像这个样子(fn [parameters*] body)。让我们来看个例子。

给定一个单词列表，用高阶函数构造一个计算单词长度的列表。比如说我们有一组人名：

```
user=> (def people ["Lea", "Han Solo"])
#'user/people
```

可以像下面这样计算一个单词长度：

```
user=> (count "Lea")
3
```

可以像下面这样计算一组人名的长度，返回一个长度的列表：

```
user=> (map count people)
(3 8)
```

这些概念之前已经见过了。其中count是高阶函数。在Clojure中，这个概念很容易理解，因为一个函数就是一个列表，和其他任何列表元素一样。还可以用同样的方式来计算人名长度的两倍长度，像下面这样：

```
user=> (defn twice-count [w] (* 2 (count w)))
#'user/twice-count
user=> (twice-count "Lando")
10
user=> (map twice-count people)
(6 16)
```

由于这个函数非常简单，因此完全可以改写成匿名函数，像下面这样：

```
user=> (map (fn [w] (* 2 (count w))) people)
(6 16)
```

还可以使用更简短的形式：

```
user=> (map #(* 2 (count %)) people)
(6 16)
```

在这个简写形式中，#定义了一个匿名函数，而%则被绑定到序列中的每个元素上。#叫做宏读取器（reader macro）。

利用匿名函数可以创建不需要名字的函数。在别的语言中你已经见过它们了。下面演示可以在集合上使用的高阶函数。所有这些函数都使用一个共同的向量v：

```
user=> (def v [3 1 2])
#'user/v
```

在下面的例子里，我们会用到这个列表及一些匿名函数。

1. apply

apply函数在参数列表上应用指定函数。(apply f '(x y))类似(f x y)，例如：

```
user=> (apply + v)
6
user=> (apply max v)
3
```

2. filter

filter函数与Ruby中的find_all相像。给定一个测试函数它会返回所有通过测试的元素的序列。比如说，要获取奇数元素或小于3的元素，可以这么做：

```
user=> (filter odd? v)
(3 1)
user=> (filter #(< % 3) v)
(1 2)
```

在深入学习Clojure序列时我们会进一步了解一些匿名函数。现在让我们稍事休息，来看看Clojure的创造者Rich Hichkey有什么要说的。

7.2.8　Rich Hickey访谈录

Rich Hichkey为本书的读者们回答了一些问题。他特别感兴趣的一个话题是：与其他Lisp版本相比，为什么这个版本的Lisp会获得更广泛的成功。所以这个访谈录所涵盖的话题范围会比其他访谈录要更宽泛一些。我希望你也能像我一样喜欢他的回答。

Bruce：你为什么要开发Clojure?

Rich：我不过是一个实践者，希望能在像JVM和CLR这样的业界标准平台上找到一种采用主流函数式且可扩展的动态语言，而且它能很好地支持并发，但是我没找到。

Bruce：你最喜欢它哪一点呢?

Rich：我喜欢它的数据结构和类库的抽象性以及简单性。也许这是两件事，但它们是相关联的。

Bruce：如果能让时光倒流，你想改变哪些特性?

Rich：我会尝试数值类型的不同实现。在JVM上装箱的数值绝对是一个悲剧。这也是我目前正在积极努力的一个领域。

Bruce：你见到过用Clojure解决的最有趣的问题是什么?

Rich：我想Flightcaster[①]（实时预测航班延误的服务）利用了Clojure的很多方面，比如利用宏的语法抽象性创建用于机器学习和统计推理的DSL，利用Java互操作能力来使用像Hadoop和

① 参见http://www.infoq.com/articles/flightcaster-clojure-rails。——原书注

Cascading这样的基础设施。

Bruce：很多Lisp方言在获取更广泛的成功方面都败下阵来，Clojure如何能成功？

Rich：这是个非常重要的问题！我不否认主要Lisp方言（Scheme和Common Lisp）已经完成了它们的使命。Scheme的目的是通过一个很精炼的语言来捕捉计算的本质，而Common Lisp则致力于标准化科研中所用到的各种Lisp方言。但它们都没能够作为一种实用工具成为业界开发者所使用的通用编程语言，实际上这也不是它们的设计目标。

另一方面，Clojure是设计用来作为一种实用工具，为业界的开发者所使用的一种通用生产性编程语言，为此在传统的Lisp上加入了这些新的目标。它更适于团队开发，擅长和其他语言相互协作，而且还解决了一些传统的Lisp问题。

Bruce：为什么Clojure在团队开发中表现更好？

Rich：某种意义上说，有些Lisp最关心的是如何挖掘个体开发者的最大力量，而Clojure认为开发是一项团队活动。例如，它不支持用户自定义的宏读取器，因为这样做会导致代码被写成一堆互不兼容的微方言。

Bruce：为什么Clojure选择运行在现有的虚拟机上？

Rich：现如今，全世界已经积累了大量有价值的、使用其他语言编写的代码，而这种情况在Lisp刚被开发出来的那个年代还没出现。因此在今天能够调用其他语言和被其他语言调用的能力必不可少，尤其是在JVM和CLR①上。

剥离宿主操作系统的标准多语言平台，在Lisp被发明的那个年代几乎不存在。整个业界比以前大出好几个量级，事实标准也已经诞生。从技术角度讲，类似复杂、精密的垃圾收集器，以及HotSpot这样的动态编译器，都是支持重用核心技术的技术层工具。因此，Clojure更强调语言在平台上（language-on-platform），而非语言即平台（language-is-platform）。

Bruce：你说的没错，但是这门Lisp方言如何才能更平易近人？

Rich：可以改进的地方很多。比如说，我们想过要处理括号“问题”。Lisp程序员深知代码即数据的价值，但是轻易放弃那些被括号阻挡住的开发者是不对的。我认为从`foo(bar, baz)`转到`(foo bar baz)`对开发者来说并不是很困难。不过我的确花了很多功夫来观察老Lisp代码中括号的使用，找寻是否存在改进的机会，结果确实有改进的空间。老Lisp的一切都用括号，而Clojure已有很大的进步。老Lisp里括号简直是太多了。Clojure选择了相反的方式，尽量去掉成对儿的括号，虽然这使得编写宏的工作变得稍微困难了一些，但却方便了使用者。

更少的括号，几乎不存在括号重载，这二者使得Clojure远比老Lisp更容易审查、阅读和理解。事实上类似`((AType)athing).amethod()`这样以两个括号打头的可怕代码在Java中比在Clojure中要更常见。

7.2.9　第一天我们学到了什么

Clojure是在JVM上的函数式语言。与Scala和Erlang类似，这种Lisp方言不是纯函数式语言。

① CLR（Common Language Runtime，公共语言运行时），.NET平台的虚拟机。——原书注

它允许有限的特殊情况。与其他Lisp方言不同的是，Clojure增加了一些语法，映射表用大括号，而向量用方括号。可以用逗号作为空白符，在有些地方还能省略掉括号。

我们学习了使用Clojure基本形式。较为简单的形式包括布尔值、字符、数字、关键字和字符串。我们还了解了各种集合。列表和向量是有序集合，向量是为随机访问优化的，而列表是为顺序遍历优化的。set是无序集合，映射表则是键值对。

我们定义了一些函数，通过提供函数名、参数列表、函数体以及一个文档字符串（可选）来完成定义。接下来在介绍绑定时用到了解构，解构可以将任意形参绑定到实参的任意部分上。这些特性令人联想起Prolog和Erlang。最后，我们定义了一些匿名函数并通过`map`函数使用它们来遍历集合。

在第二天，我们会看看Clojure的递归，这也是绝大部分函数式语言中的一个基本组成部分。我们还会介绍序列和延迟计算，它们作为Clojure模型基石的一部分，帮助实现了集合之上的强大的公共抽象层。

现在，我们先休息一下，练习一下前面所学的内容。

7.2.10 第一天自习

Clojure是一门新的编程语言，但是你仍会发现其社区异常活跃且增长迅速。在为本书进行研究时，Clojure社区是我找到的最棒的社区之一。

找

- 使用Clojure序列的例子。
- Clojure函数的正式定义。
- 用于在你的环境中启动repl的脚本。

做

- 实现一个函数`(big st n)`，当字符串`st`长度不超过`n`个字符时返回`true`。
- 实现一个函数`(collection-type col)`，根据给定集合`col`的类型返回`:list`，`:map`或`:vector`。

7.3 第二天：Yoda与原力

7

作为一名绝地大师，Yoda训练他的学徒们使用和理解一切生命体的统一体现——原力。在这一节中，我们将开始学习Clojure中最基本的概念。我们会谈到序列、统一Clojure集合以及将这些集合与Java集合连接的抽象层。我们还会介绍延迟计算，按需计算序列成员的即时策略。接着我们会探究所有Lisp原力之所在，那个神秘的语言特性：宏。

7.3.1 用`loop`和`recur`递归

你在本书的其他语言那儿已经学到过，函数式语言依赖递归而非遍历。下面是一个计算向量大小的递归程序：

```
(defn size [v]
    (if (empty? v)
            0
            (inc (size (rest v)))))

(size [1 2 3])
```

理解起来并不困难，空列表大小为零，非空列表比其尾部多一。我们已经见过其他语言的类似方案。

栈会增长，因此递归算法会持续消耗内存直至内存耗尽。函数式语言通过使用尾递归优化技术规避这个问题。受到JVM限制，Clojure并不支持隐式的尾递归优化。你必须显式地使用`loop`和`recur`进行递归，可以把`loop`看作是`let`语句。

```
(loop [x x-initial-value, y y-initial-value] (do-something-with x y))
```

初始时，对于给定向量，`loop`会将向量中偶数位变量绑定到奇数位的值上。事实上，如果不声明`recur`，`loop`和`let`效果完全一样：

```
user=> (loop [x 1] x)
1
```

函数`recur`会再次调用循环并传入新值。

用`recur`重构`size`函数：

```
(defn size [v]
        (loop [l v, c 0]
        (if (empty? l)
                c
                (recur (rest l) (inc c)))))
```

在`size`的第二个版本中，使用了经过尾递归优化的`loop`和`recur`。由于并不真正返回结果值，因此我们在变量中保存结果，这个变量称为*累加器*。在这个例子里，`c`用于计数。

这个版本会像经过尾递归优化一样工作，但是我们却为此新增了几行看起来非常糟糕的代码。有时，JVM就像一把双刃剑。如果你需要交互，那就必须把问题处理掉。不过由于这个函数已经加入到集合的基础API中，你不会经常需要用到`recur`。此外，Clojure也提供了一些优秀的递归替代技术，包括后面会讲到的延迟序列。

第二天的最乏味的部分到这里就介绍完了，下面转向更令人愉快的事物。我们将要了解序列的一些特性，而正是这些特性使得Clojure如此特别。

7.3.2　序列

序列是与具体实现无关的抽象层，囊括了Clojure系统里各式的容器。序列封装了所有Clojure集合（set、映射表、向量，等等）、字符串，甚至包括文件系统结构（流、目录）。它们也为Java容器提供了公共抽象，包括Java集合、数组以及字符串。一般来说，只要支持函数`first`、`rest`和`cons`，就可以用序列封装起来。

使用向量时，Clojure有时会在命令行中返回列表：

```
user=> [1 2 3]
[1 2 3]
user=> (rest [1 2 3])
(2 3)
```

注意，开始用的是向量，但结果并不是列表。repl实际上是返回序列作为响应。这意味着所有集合都能以一种通用方式对待。让我们来看看公共序列函数库，它的丰富与强大，用一节的内容难以描述，但我会试着让你了解都有哪些可用的函数，下面简单介绍一下用于修改、测试和创建序列的函数。

1. 测试

如果要测试序列，可以用判定函数。它接受序列和一个测试函数并返回布尔值。如果测试函数对所有序列元素测试结果都是真，every?会返回true：

```
user=> (every? number? [1 2 3 :four])
false
```

这里有一个元素不是数字。只要序列中有元素测试为真，some函数就会返回true：[1]

```
(some nil? [1 2 nil])
true
```

其中一个元素为nil。not-every?和not-any?则反过来：

```
user=> (not-every? odd? [1 3 5])
false
user=> (not-any? number? [:one :two :three])
true
```

下面来看看修改序列的函数。

2. 修改序列

序列函数库中包含一系列用于以各种方式转换序列的函数。你已经见过filter函数。要提取出长度大于4的单词，用下面这段代码：

```
user=> (def words ["luke" "chewie" "han" "lando"])
#'user/words
user=> (filter  (fn [word] (> (count word) 4)) words)
("chewie" "lando")
```

另外你也已经见过map函数，它对集合所有元素调用同一函数并返回其结果。用向量的所有元素创建一个由其平方数组成的序列：

```
user=> (map (fn [x] (* x x)) [1 1 2 3 5])
(1 1 4 9 25)
```

列表解析结合了映射和过滤，如你在Erlang和Scala中见到的一样。列表解析能组合多个列表和过滤器，穷举列表之间所有可能的组合，然后对其使用过滤器。我们先看个简单例子。有两个列表，colors和toys：

7

[1] 更准确地说，some返回第1个不是nil或false的值。例如，(some first[[][1]])会返回1。——原书注

```
user=> (def colors ["red" "blue"])
#'user/colors
user=> (def toys ["block" "car"])
#'user/toys
```

列表解析可以结合函数使用，类似于映射：

```
user=> (for [x colors] (str "I like " x))
("I like red" "I like blue")
```

[x colors]将x绑定到来自colors列表的元素上。而(str "I like " x)则是应用于colors中的每个x的函数。当你绑定多于一个列表时，事情会变得更有趣：

```
user=> (for [x colors, y toys] (str "I like " x " " y "s"))
("I like red blocks" "I like red cars"
 "I like blue blocks" "I like blue cars")
```

列表解析创建出两个列表之间所有可能的组合。绑定时还能通过:when关键词来过滤：

```
user=> (defn small-word? [w] (< (count w) 4))
#'user/small-word?
user=> (for [x colors, y toys, :when (small-word? y)]
            (str "I like " x " " y "s"))
("I like red cars" "I like blue cars")
```

我们编写了一个叫small-word?的过滤器。定义任何长度小于4个字符的单词算是small-word。通过:when (smallword? y)对y应用了过滤器small-word?。这样一来就能得到所有可能的组合(x, y)，其中x是colors的成员，y是toys的成员，且y的长度小于4个字符。代码包含的信息量很密集，但是却清晰明了。这是理想的组合，让我们继续吧。

你已经在Erlang、Scala和Ruby中见过了foldl、foldleft和inject。在Lisp中，它们的等价的是reduce函数。求和或求阶乘可以这样写：

```
user=> (reduce + [1 2 3 4])
10
user=> (reduce * [1 2 3 4 5])
120
```

可以这样进行列表排序：

```
user=> (sort [3 1 2 4])
(1 2 3 4)
```

也可以按函数结果进行排序：

```
user=> (defn abs [x] (if (< x 0) (- x) x))
#'user/abs
user=> (sort-by abs [-1 -4 3 2])
(-1 2 3 -4)
```

这里定义了一个名为abs的函数来计算绝对值，接着在排序中使用了它。以上就是Clojure最重要的序列转换函数。接下来我们介绍创建序列的函数，在进行之前，你必须变得稍微“迟钝”一点儿。

7.3.3　延迟计算

在数学中，无限数列通常很容易描述。在函数式语言中，也常能享受到同样的益处，不过不能真去计算一个无穷序列，这个问题的解决办法就是延迟计算。通过使用这种策略，Clojure的序列函数库只会在一个值真正被使用时才去计算它。事实上，绝大多数序列都是延迟计算的。让我们简单了解一下创建有穷序列的过程，然后再讲解延迟序列。

1. 用range创建有穷序列

与Ruby不同，Clojure支持range函数。range能创建一个序列：

```
user=> (range 1 10)
(1 2 3 4 5 6 7 8 9)
```

注意，上限不包含在区间内。序列中不包含10。还可以指定任意步长：

```
user=> (range 1 10 3)
(1 4 7)
```

如果不指定步长，也可以不指定下限：

```
user=> (range 10)
(0 1 2 3 4 5 6 7 8 9)
```

零是默认下限。用range创建出来的序列都是有穷的。那如果想创建一个无上限的序列要如何做？那是一个无穷序列。下面来看看如何创建无穷序列。

2. 无穷序列和take

先从最基本的无穷序列开始，即不断重复同一元素的序列，这可以用(repeat 1)得到。如果在repl中尝试这个命令的话，屏幕会不停地输出1，直到结束进程为止。很明显，我们需要某种方法来只获取其中一个有限的子集。这就需要使用函数take：

```
user=> (take 3 (repeat "Use the Force, Luke"))
("Use the Force, Luke" "Use the Force, Luke" "Use the Force, Luke")
```

如上，我们创建了一个不断重复字符串"Use the Force, Luke"的无穷序列，接着取出了前三个元素。还可以用函数cycle来重复列表中的元素：

```
user=> (take 5 (cycle [:lather :rinse :repeat]))
(:lather :rinse :repeat :lather :rinse)
```

我们从向量[:lather :rinse :repeat]的循环中取出前5个元素。除此以外，我们还能丢弃序列中的前几个元素：

```
user=> (take 5 (drop 2 (cycle [:lather :rinse :repeat])))
(:repeat :lather :rinse :repeat :lather)
```

从内向外执行，第一步还是创建循环，然后丢弃前2个元素，接着取出紧跟在后面的5个元素。但也不是必须从内向外执行。可以使用新的从左向右操作符 (->>)，将每个函数分别应用于一个结果：

```
user=> (->> [:lather :rinse :repeat] (cycle) (drop 2) (take 5))
(:repeat :lather :rinse :repeat :lather)
```

就这样，先创建一个向量，转入函数cycle创建序列，然后丢弃2个元素，接着取出5个元素。

7

有时候，从左向右的代码读起来还是更容易一些。如果想在单词间添加分隔符，可以用函数 interpose：

```
user=> (take 5 (interpose :and (cycle [:lather :rinse :repeat])))
(:lather :and :rinse :and :repeat)
```

我们把关键词:and插入到无穷序列的所有元素之间。可以把这个函数当作是Ruby中join的通用版本。如果想执行一个插入，而插入用的元素取自一个序列该怎么办？那就要用到函数 interleave：

```
user=> (take 20 (interleave (cycle (range 2)) (cycle (range 3))))
(0 0 1 1 0 2 1 0 0 1 1 2 0 0 1 1 0 2 1 0)
```

我们将两个无穷序列(cycle (range 2))和(cycle (range 3))交错。接着，取出前20个元素。结果正如你看到的，偶数位是(0 1 0 1 0 1 0 1 0 1)，奇数位是(0 1 2 0 1 2 0 1 2 0)，太漂亮了。

iterate函数提供另一种创建序列的方法。看看下面的例子：

```
user=> (take 5 (iterate inc 1))
(1 2 3 4 5)
user=> (take 5 (iterate dec 0))
(0 -1 -2 -3 -4)
```

iterate函数接受一个函数和一个起始值作为参数。然后iterate对起始值重复调用作为参数传入的那个函数。上面例子中，分别调用了函数inc和dec。

下面是计算斐波那契数列的例子，该数列的每个数字都是前两个数值之和。对给定数值对 [a b]，可以用[b, a + b]来生成下一对。在给定前一对的条件下，可以创建匿名函数来生成下一对，像下面这样：

```
user=> (defn fib-pair [[a b]]  [b (+ a b)])
#'user/fib-pair
user=> (fib-pair [3 5])
[5 8]
```

接下来，我们会用函数iterate构建一个无穷数列。先别急着执行这行：

```
(iterate fib-pair [1 1])
```

我们用函数map提取出所有数值对的第一个元素：

```
(map
first
(iterate fib-pair [1 1]))
```

这也是个无穷数列。现在，可以提取出前5个元素：

```
user=> (take 5
         (map
         first
         (iterate fib-pair [1 1])))
(1 1 2 3 5)
```

或者也可以提取出第500个元素，像下面这样：

```
(nth (map first (iterate fib-pair [1 1])) 500)
(225... more numbers ...626)
```

性能非常出色。利用延迟序列，描述类似斐波那契数列这样的递归问题非常容易。另一个例子是阶乘：

```
user=> (defn factorial [n] (apply * (take n (iterate inc 1))))
#'user/factorial
user=> (factorial 5)
120
```

上面从无穷数列`(iterate inc 1)`中取出*n*个元素。然后用`apply *`将它们相乘。这个办法非常简单。接下来探索Clojure函数`defrecord`和`protocol`。

7.3.4 defrecord和protocol

到目前为止，我们只在较高的层面上谈论过Java集成，但是你还没有看到JVM渗透入Clojure的全貌。说到底，JVM就是与类型和接口有关。（如果你不是Java程序员，可以把类型看作是Java类，接口看作是没有实现的Java类。）为了使Clojure能很好地集成JVM，最初的实现中有大量Java。

随着Clojure速度提升并且开始证明自己是一种高效的JVM语言，人们强烈地认为应该用Clojure语言本身来实现更多的Clojure。为了做到这一点，Clojure开发者需要找到一种方法来构建平台高性能[①]的开放扩展，使得他们可以对抽象编程，而非对具体实现编程。得到的结果是`defrecord`用于类型，`protocol`用于围绕类型来组织函数。从Clojure的观点看，面向对象（OO）中最好的部分是类型和协议（比如接口），而最差的部分则是对具体实现的继承。Clojure的`defrecord`和`protocol`就是去其糟粕，取其精华。

在本书编写过程中，这些重要的语言特性仍在不断发展之中。我需要仰仗Stuart Halloway的帮助来完成一个有实际意义的实现。他是Relevance的创始人，*Programming Clojure*[Hal09]一书的作者。我们将会回顾JVM上的另外一门函数式语言Scala，还会用Clojure重写罗盘程序。让我们赶紧开始吧。

首先，我们来定义一个协议。Clojure协议类似契约。这个协议的类型能支持一组具体的函数、字段和参数。下面是一个描述`Compass`的协议：

clojure/compass.clj

```
(defprotocol Compass
  (direction [c])
  (left [c])
  (right [c]))
```

上面的协议定义了一个叫做`Compass`的抽象，并列举了`Compass`必须支持的所有函数——`direction`、`left`和`right`及其指定的参数。接下来就可以自由使用`defrecord`来实现协

① 据Bruce Tate解释，这里的平台指具体计算机。JVM有时不够快。由于`defrecord`和`protocol`被用于实现非常底层的函数，可能要每秒执行上千次。它们需要被优化，有时是通过下放到更底层的语言来实现。

议了。接下来，我们需要定义4个方向：

```
(def directions [:north :east :south :west])
```

然后我们还需要一个函数来处理转向。回忆一下，基本方向:north、:east、:south和:west分别由整数0、1、2、3表示。在初始方向上每加1就让罗盘右转90°。这样通过base/4（更准确的说是base/number-of-directions）取余就能准确计算从:west到:north的转向，如下所示：

```
(defn turn
  [base amount]
  (rem (+ base amount) (count directions)))
```

转向能用了。现在加载罗盘的代码文件，然后试试turn函数：

```
user=> (turn 1 1)
2
user=> (turn 3 1)
0
user=> (turn 2 3)
1
```

换句话说，从:east向右转一次得到:south，从:west向右转一次得到:north，从:south向右转三次得到:east。

接下来实现协议。轮到defrecord上场了。我们会一部分一部分地完成实现。首先，用defrecord声明实现，像下面这样：

```
(defrecord SimpleCompass [bearing]
  Compass
```

我们定义了一个名为SimpleCompass的新记录。它有一个字段叫做bearing。接下来实现Compass协议，从函数direction开始：

```
(direction [_] (directions bearing))
```

direction函数在directions中查找下标为bearing的元素。例如，(directions 3)返回:west。每个参数列表都有一个指向对象实例的引用（比如Ruby中的self，或者Java中的this），但是我们并没有使用它，因此在参数列表中加入了_（下划线）。接下来实现函数left和right：

```
(left [_] (SimpleCompass. (turn bearing 3)))
(right [_] (SimpleCompass. (turn bearing 1)))
```

记住，在Clojure中，我们使用的是不可变值。这意味着所有转向函数都要返回一个新的、修改好的罗盘，而不是直接修改原来的罗盘。函数left和right使用了你以前没见过的语法。(SomeType. arg)意味着调用SimpleCompass的构造函数，并绑定arg到第一个参数上。可以这样验证，在repl中输入(String. "new string")，repl会返回一个新字符串"new string"。

函数left和right都很容易。各自都返回一个新罗盘，其方位（由指定的新方位配置）用前面定义好的函数turn即可。right向右转90°一次，而left向右转90°三次。目前为止，我们得到一个实现协议Compass的类型SimpleCompass。就差一个返回字符串表示的函数了，不过

toString是java.lang.Object的方法，要像下面这样把它添加到类型里，很简单对吧。

```
Object
(toString [this] (str "[" (direction this) "]")))
```

通过实现toString方法实现了Object协议的一部分，返回一个类似SimpleCompass [:north]的字符串。

现在，类型终于完整了。下面来创建一个新罗盘：

```
user=> (def c (SimpleCompass. 0))
#'user/c
```

转向会返回一个新罗盘：

```
user=> (left c) ; 返回一个新罗盘
#:SimpleCompass{:bearing 3}

user=> c ; 原来的罗盘没有变化
#:SimpleCompass{:bearing 0}
```

注意，老罗盘没变。由于我们定义的是一个JVM类型，因此可以把所有字段都当作Java字段来访问。另外还能把类型里的字段看作Clojure映射表的关键词进行访问：

```
user=> (:bearing c)
0
```

因为这些类型用起来很像映射表，所以可以很容易地把映射表当作新类型的原型，然后逐步迭代，设计稳定后再把它们替换成类型。也可以在测试中用映射表替换各种类型作为测试桩或者模拟对象。除此外还有一些其他好处。

- 类型能和其他Clojure的并发编程结构很好地共处。在第三天中，我们会学习如何创建对Clojure对象的可变引用，同时保持事务完整性，就像关系型数据库那样。
- 我们实现了一个protocol，但并没有被局限于只使用新方法。因为我们创建的是JVM类型，这些类型可以和Java类型和接口互操作。

通过defrecord和protocol，Clojure提供了在不使用Java的前提下构建JVM本地代码的能力。这些代码可以与JVM上的其他类型全面交互，包括Java类型和接口。你可以使用它们继承Java类型或者实现Java接口。Java类型也可以构建在Clojure类型之上。当然，这不是Java互操作的全部内容，但却是非常重要的一部分。现在你已经学会了如何扩展Java，接下来让我们学习如何通过宏来扩展Clojure语言自身。

7

7.3.5　宏

这一节中，我们会引用Io那章的内容。我们已经用Io实现了Ruby的unless，即3.3节的Messages。其形式为(unless test form1)。如果test为false则函数执行form1。这个函数不能就这么简单地直接设计出来，因为每个参数都会执行：

```
user=> ; 坏掉的unless
```

```
user=> (defn unless [test body] (if (not test) body))
#'user/unless
user=> (unless true (println "Danger, danger Will Robinson"))
Danger, danger Will Robinson
nil
```

Io一章中我们讨论过这个问题。绝大多数语言会先执行参数，再把结果放到调用栈上。但是这个例子里，我们不希望对代码块进行求值，除非条件为假。在Io中，语言通过延迟执行`unless`消息规避了这个问题。在Lisp中，我们可以使用宏。当输入`(unless test body)`时，我们想让Lisp将其翻译成`(if (not test) body)`，这时宏就派上用场了。

Clojure程序的执行分为两个阶段。宏展开（macro expansion）阶段将语言里的所有宏翻译成其展开形式。你可以用命令`macroexpand`观察宏展开。我们已经用过几个宏了，它们都叫做*宏读取器*。分号（;）表示注释，单引号（'）表示引用，而数学符号（#）则表示匿名函数。为了避免早于预期执行，在想展开的表达式前面加上一个引号：

```
user=> (macroexpand ''something-we-do-not-want-interpreted)
(quote something-we-do-not-want-interpreted)
user=> (macroexpand '#(count %))
(fn* [p1__97] (count p1__97))
```

这些就是宏。一般来说，宏展开允许你把代码当作列表来处理。如果不想立即执行一个函数，那就把它引起来。Clojure会完整地替换参数。我们的`unless`看起来会像这样：

```
user=> (defmacro unless [test body]
         (list 'if (list 'not test) body))
#'user/unless
```

注意，Clojure在替换`test`和`body`时不会对它们进行求值，但是必须把`if`和`not`引起来，而且还要把它们打包到列表中。我们创建了一个代码列表，其形式与Clojure将会执行的形式完全相同。可以对它用`macroexpand`进行宏展开：

```
user=> (macroexpand '(unless condition body))
(if (not condition) body)
```

还能像下面这样执行：

```
user=> (unless true (println "No more danger, Will."))
nil
user=> (unless false (println "Now, THIS is The FORCE."))
Now, THIS is The FORCE.
nil
```

我们修改了语言的基本定义，增加了自己的控制结构，而不需要语言的设计者去增加他们自己的关键字。宏扩展恐怕是Lisp中最强大的特性了，而且没有多少语言能做到这一点。秘诀就在于用数据来表示代码，而不只是字符串。代码本身就已是高阶数据结构了。

让我们来总结一下第二天。今天干货很足。我们应该暂停一下来运用所学知识。

7.3.6　第二天我们学到了什么

又是充实的一天。收获了这么多抽象技术，你的锦囊一定扩充了不少，现在让我们来回顾一下。

首先，我们学习了如何使用递归。由于JVM不支持尾递归优化技术，所以我们不得不使用`loop`和`recur`。虽然其语法本身看起来比较有颠覆性，但这种循环编程结构可以实现许多算法，而且通常会用递归调用来实现。

我们还使用了序列。通过它们，Clojure封装了所有对集合的访问。利用公共函数库，在处理集合时可以使用共有的策略。我们使用了不同的函数来修改、转换、搜索序列。高阶函数为序列函数库增添了力量和简洁性。

有了延迟序列，我们就能够为序列增加另一个强大的层次。延迟序列简化了算法，它们还提供了延迟计算，从而潜在地大幅提升性能，降低耦合。

接下来，我们花了一些时间来实现类型。有了`defrecord`和`protocol`，我们就能够实现类型，JVM上的"一等公民"。

最后，我们使用宏给语言添加了特性。我们了解到有一个步骤名为宏展开，它发生在Clojure实现或解释代码之前。通过在宏展开中使用函数`if`，我们实现了`unless`。

这些知识足够消化上一阵子的了。现在花些时间来实践一下所学内容吧。

7.3.7　第二天自习

今天的学习内容里塞满了Clojure语言中最尖端、最强大的元素。花些时间来探究和理解这些元素吧。

找

- Clojure中一些常用宏的实现。
- 一个自定义延迟序列的例子。
- `defrecord`和`protocol`目前的状态。（在编写本书时，这些特性还在开发中。）

做

- 用宏实现一个包含`else`条件的`unless`。
- 编写一个类型用`defrecord`实现一个协议。

7.4　第三天：一瞥魔鬼

7

在《星球大战》中，Yoda首先发现了Darth Vader心中的邪念。对于Clojure，Rich Hickey已经找出了在开发面向对象并发系统时带来麻烦的核心问题。我们经常说，可变状态是潜伏在面对对象程序心脏里的魔鬼。前几章展示过几种处理可变状态的不同方法。Io和Scala使用基于actor的模型并提供了不可变编程结构，给予程序员在不依赖可变状态下解决这些问题的力量。Erlang提供了actor和轻量级进程，以及一个支持高效监控和通讯的虚拟机，带来了前所未有的可靠性。Clojure的并发之路又不同。它使用了STM（software transactional memory，软件事务内存）。在这一节里，我们会学习STM以及几种多线程应用里共享状态的工具。

7.4.1　引用和事务内存

数据库使用事务来保证数据完整性。现代数据库至少使用两种类型的并发控制。锁能够防止两个竞争事务同时访问同一行记录。多版本支持每个事务拥有其私有的数据。如果任何一个事务妨碍了其他事务，数据库引擎就会直接重新执行该事务。

像Java这样的语言使用锁保护一个线程的资源免遭其他竞争线程的破坏。锁基本上把并发控制的负担交给了程序员。我们很快意识到这个负担实在太重，难以承受。

像Clojure这样的语言使用STM。其策略是使用多版本维持一致性和完整性。不同于Scala、Ruby或是Io，在Clojure中如果想修改一个引用的状态，必须在事务内进行操作。让我们来看看它是如何工作的。

引用

在Clojure中，`ref`（引用的简写）是一份封装好的数据。所有访问必须遵守一定的规则。这种情况下，规则就是要支持STM，即不能在事务之外修改引用。为了观察它是如何工作的，我们先创建一个引用：

```
user=> (ref "Attack of the Clones")
#<Ref@ffdadcd: "Attack of the Clones">
```

这么写没什么意思。应该将引用赋给一个值，像下面这样：

```
user=> (def movie (ref "Star Wars"))
#'user/movie
```

可以获取引用，像下面这样：

```
user=> movie
#<Ref@579d75ee: "Star Wars">
```

但是我们真正关心的是引用里的值。使用`deref`，如下所示：

```
user=> (deref movie)
"Star Wars"
```

或者，也可以使用`deref`的简写形式：

```
user=> @movie
"Star Wars"
```

这样看起来好多了。现在能很容易地访问引用里的值。我们还没试过修改引用的状态，让我们来试试。在Clojure中，我们传入一个负责修改值的函数，解引用后的`ref`则作为函数的第一个参数被传入：

```
user=> (alter movie str ": The Empire Strikes Back")
java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)
```

如你所见，只能在事务里修改状态。这要用到函数`dosync`。修改引用的首选方式是使用转换函数来修改它，像下面这样：

```
user=> (dosync (alter movie str ": The Empire Strikes Back"))
"Star Wars: The Empire Strikes Back"
```

还可以用`ref-set`设置初始值：

```
user=> (dosync (ref-set movie "Star Wars: The Revenge of the Sith"))
"Star Wars: The Revenge of the Sith"
```

可以看到引用已被修改：

```
user=> @movie
"Star Wars: The Revenge of the Sith"
```

和我们期望的一样，引用发生了变化。尽管用这种方式修改可变变量（mutable variable）看起来挺痛苦的，但Clojure在这里的强制政策会免除以后的很多麻烦。我们知道以这种方式运转的程序绝对能正确运行，即便是考虑到竞争条件或者死锁。大部分代码用函数式编程范型，而对那些最能通过可变性受益的问题，则使用STM。

7.4.2　使用原子

如果想保证单个引用的线程安全，且不需要与其他活动协调，那么就可以用原子。这类数据元素可以在事务的上下文之外被修改。和引用类似，Clojure原子（atom）也是封装好的状态。我们来试试看。先创建一个原子：

```
user=> (atom "Split at your own risk.")
#<Atom@53f64158: "Split at your own risk.">
```

接着，绑定原子：

```
user=> (def danger (atom "Split at your own risk."))
#'user/danger
user=> danger
#<Atom@3a56860b: "Split at your own risk.">
user=> @danger
"Split at your own risk."
```

可以用`reset!`重新绑定`danger`到一个新值上：

```
user=> (reset! danger "Split with impunity")
"Split with impunity"
user=> danger
#<Atom@455fc40c: "Split with impunity">
user=> @danger
"Split with impunity"
```

`reset!`替换了整个原子，不过首选的方法是提供一个函数来变换原子。如果要修改一个很大的向量，可以使用`swap!`原地修改原子，如下所示：

```
user=> (def top-sellers (atom []))
#'user/top-sellers
user=> (swap! top-sellers conj {:title "Seven Languages in Seven Weeks", :author "Tate"})
[{:title "Seven Languages in Seven Weeks", :author "Tate"}]
user=> (swap! top-sellers conj {:title "Programming Clojure" :author "Halloway"})
[{:title "Seven Languages in Seven Weeks", :author "Tate"}
 {:title "Programming Clojure", :author "Halloway"}]
```

和引用一样，一次创建好值，然后用`swap!`来修改。让我们来看一些实用例子。

构建原子缓存

现在，你已经见过引用和原子。等我们学习Haskell时你还会见到同样的通用编程哲学。将一小块状态封装成包，而后用函数来修改。修改引用需要事务，原子不需要。接下来构建一个简单的原子缓存。这个问题很适合用原子解决。用散列来关联名字和值就行。要感谢Stuart Halloway of Relevance[①]提供这个例子，它是一家提供Clojure培训和咨询的公司。

我们需要先创建一个缓存，然后创建向缓存里添加元素的函数以及从缓存里删除元素的函数。首先是创建缓存：

clojure/atomcache.clj

```
(defn create
  []
  (atom {}))
```

这里只是简单地创建了一个原子，由这个类的用户来绑定它。接下来，需要能根据缓存的键获取元素：

```
(defn get
  [cache key]
  (@cache key))
```

函数以缓存和键作为参数。缓存是一个原子，因此需对其解引用，然后返回与键相关联的元素。最后，还需要能往缓存里放入元素的函数：

```
(defn put
  ([cache value-map]
     (swap! cache merge value-map))
  ([cache key value]
     (swap! cache assoc key value)))
```

我们定义了两个不同的函数put。第一个版本用merge，使我们能够将一个映射表中的所有关联全部加入进缓存。第二个版本使用assoc来添加一对键和值。下面使用缓存。我们先添加一个元素到缓存中，然后再返回它：

```
(def ac (create))
(put ac :quote "I'm your father, Luke.")
(println (str "Cached item: " (get ac :quote)))
```

输出是：

```
Cached item: I'm your father, Luke.
```

同步情况下，原子和引用都是处理可变状态既简单又安全的方法。在下面的小节里，我们会看几个异步的例子。

7.4.3　使用代理

像atom一样，代理也是封装起来的一份数据。与Io中的future相似，解引用后的代理会一直

① 参见http://www.thinkrelevance.com。——原书注

阻塞直到有值可用。使用者可以用函数异步修改数据，而更新会在另一个线程中发生。每次只有一个函数能修改代理的状态。

来试试看。定义一个函数叫twice，它将传入值放大两倍：

```
user=> (defn twice [x] (* 2 x))
#'user/twice
```

接下来，定义一个叫tribbles的代理，初始值为1：

```
user=> (def tribbles (agent 1))
#'user/tribbles
```

现在，可以通过给代理发送一个值来修改tribbles：

```
user=> (send tribbles twice)
#<Agent@554d7745: 1>
```

这个函数会在另一个线程上执行。我们来取出代理的值：

```
user=> @tribbles
2
```

从引用、代理、或原子中读取值永远都不会锁定也永远不会阻塞。读应该尽可能快，有了正确抽象的封装，就能做到这点。用下面这个函数，你就能看出从代理读取到的各个值之间的区别了：

```
user=> (defn slow-twice [x]
         (do
             (Thread/sleep 5000)
             (* 2 x)))
#'user/slow-twice
user=> @tribbles
2
user=> (send tribbles slow-twice)
#<Agent@554d7745: 16>
user=> @tribbles
2
user=> ; 5秒后再执行
user=> @tribbles
4
```

别过分纠结于语法。(Thread/sleep 5000)就是调用Java中Thread的sleep方法。现在专注到代理的值上。

我们定义了一个需要5秒钟才能完成的慢节奏版twice函数。这样就有足够时间观察repl中@tribbles随时间推移的变化。

所以，你一定能得到一个tribbles值。但在你的线程上有可能拿不到最新的修改。如果需要确保在自己的线程里得到最新值，可以调用(await tribbles)或者(awaitfor timeout tribbles)，其中timeout是以毫秒为单位的超时时间。记住，在处理完来自线程的动作之前，await和awaitfor会一直阻塞。但这里并没提到其他线程可能要求该线程所做的事情。你无法得到最新的值。因为Clojure工具涉及使用快照，其值是瞬时的，并且很有可能立即过期。这正是版本数据库进行快速并发控制的方式。

7.4.4　future

在Java里，可以直接启动线程来完成指定任务。当然，你可以以这种方式通过Java集成来启动一个线程，不过总有更好的办法。比方说你想创建一个线程来处理复杂计算，可以用代理。又或者，如果你想开始计算某个值，但不想等待结果。像在Io中一样，可以使用future。我们来看下面的示例。

首先创建一个future，这个future立即返回一个引用：

```
user=> (def finer-things (future (Thread/sleep 5000) "take time"))
#'user/finer-things
user=> @finer-things
"take time"
```

根据你打字的速度，你可能需要等一下结果才返回。future接受一或多个表达式，并返回最后一个表达式的值。future在另外一个线程中启动。如果对其解引用，则值可用之前，future会一直阻塞。

所以，future是允许在计算完成前异步返回的并发编程结构。用future可以让多个需要长时间运行的函数并行执行。

7.4.5　还差什么

Clojure是一种Lisp方言，而Lisp本身就是一种内容极其丰富的语言。Clojure基于JVM，其发展历程已经超过10年。这门语言还混入了一些新的、强大的概念，要用一本书的其中一章来完整介绍Clojure是不可能的。还有一些你应该了解的内容。

1. 元数据

有时，你会很高兴能在类型上关联一些元数据。Clojure允许在符号和集合上附加并访问元数据。`(with-meta value metadata)`返回一个与`metadata`关联的新的`value`，通常是用映射表实现的。

2. Java集成

Clojure有非常出色的Java集成。我们零散地提了一些Java集成的内容，然后还创建了一个JVM上的类型。但是完全没有用到已有的Java类库。我们也没全面介绍Java兼容形式，举个例子，`(.toUpperCase "Fred")`会调用字符串`"Fred"`的成员函数`toUpperCase`。

3. 多重方法

面向对象语言只允许一种组织行为和数据的方式。Clojure提供多重方法（multimethods）允许你创建自己的代码组织方式。你可以把一个库的所有函数和一个类型关联起来，也可以用多重方法来实现多态，依据类型、元数据、参数甚至是属性来进行方法调度（method dispatch）。这个概念很强大、很灵活。比如说，完全可以实现Java风格的继承、原型继承，或者某种完全不同的东西。

4. 线程状态

Clojure为各种并发模型提供了原子、引用和代理。有时，需要在每个线程实例中分别存储数据。Clojure提供了`vars`，可以非常容易地完成这个工作。举个例子，`(binding [name`

"value"] ...)会将name绑定到"value"上，且仅对当前线程绑定。

7.4.6 第三天我们学到了什么

今天，我们浏览了几种并发结构，介绍了几种有趣的并发结构。

引用支持在实现可变状态的同时还保持各线程间的一致性。我们使用的是STM，即软件事务内存。从我们的角度来看，就是要把所有对引用的修改置于事务中，表示为使用函数dosync。

接下来，我们使用了原子，一种轻量级的并发结构，保护更少但使用模型更简单。我们在事务之外修改了一个原子。

最后，我们用代理实现了一个池，它可用于执行长时间计算。代理与Io的actor不同，Clojure可以用任意函数来修改代理的值，代理也会及时返回快照，一个随时都有可能被改变的值。

7.4.7 第三天自习

在第二天里，我们重点介绍高阶抽象编程。第三天我们迎来了Clojure中的并发编程结构。在以下的练习中，你将会运用所学内容。

找

- 一个队列实现，队列为空时阻塞并等待新的元素加入。

做

- 使用引用在内存中创建一组账户的向量，实现用于修改账户余额的借贷函数debit和credit。

接下来，我会描述一个称之为理发师问题（sleeping barber）的题目，它是由Edsger Dijkstra于1965年提出的，特点如下：

- 理发店接待顾客；
- 顾客到达理发店的时间间隔随机，范围10~30毫秒；
- 理发店的等待室里有三把椅子；
- 理发店只有一位理发师和一把理发椅；
- 当理发椅为空时，一位顾客坐上去，叫醒理发师，然后理发；
- 如果所有椅子都被占用，新来的顾客就会离开；
- 理发需要20毫秒；
- 完成理发后，顾客会起身离开。

实现一个多线程程序来决定一个理发师在10秒内可以为多少位顾客理发。

7.5 趁热打铁

Clojure结合了Lisp方言的力量和JVM的便利。从JVM这边，Clojure受益于已有社区、部署平台和代码库。作为Lisp方言，Clojure也有相应的优势和劣势。

7.5.1　Lisp悖论

Clojure可能是本书中最强大灵活的语言。多重方法支持多种编程范型的代码，而宏允许你动态地重定义语言。这本书里再没有第二种语言能提供如此强大的组合。这种灵活性已经被证明是一种不可思议的力量。在《黑客与画家》一书中，Graham回顾了一个创业公司如何利用Lisp发挥出其他供应商难以企及的生产力的故事。一些新兴顾问也采取同样的做法，打赌Clojure将会提供其他语言无法比拟的生产力和质量优势。

Lisp的灵活性也可以成为其弱点所在。宏展开在专家手里是强大的特性，但如果考虑不周就很可能导致严重的灾难。同样，也只有最熟练的程序员才可以不费吹灰之力就在些许Lisp代码中使用许多强大的抽象。

为了能正确地评估Clojure，你需要了解Lisp，同时也包括Java系统其他独特的方面，以及其语言中新的独有特性。让我们来更深入地探讨一下Clojure的核心优势。

7.5.2　核心优势

成为下一个Java虚拟机上最流行的语言，Clojure属于这一目标中为数不多的竞争语言之一。它有很多理由成为一个强有力的候选者。

1. 优秀的Lisp方言

Tim Bray，编程语言专家和超级博主，在*Eleven Theses on Clojure*[①]一文中称Clojure是一种优秀的Lisp方言，事实上，他称Clojure为“有史以来最好的Lisp方言”。我同意Clojure是一种优秀的Lisp方言。

在本章中，你已经读过了Rich Hickey认为的“是什么让Clojure成为一个优秀的Lisp方言”的讨论，概括如下。

- *减少括号*。通过开辟一丁点儿新语法，Clojure改进了可读性，这包括向量用方括号，映射表用大括号和set所使用的字符组合。
- *生态系统*。Lisp的许多方言都在一件事上有所妥协，即提供所有方言都可用的支持和类库。讽刺的是，再增加一种方言却又能改善这一问题。建于JVM之上使得Clojure可以充分受益于Java语言中大量优雅的类库集合。
- *克制*。通过实践克制并限制Clojure语法以避免宏读取器，Hickey限制了Clojure的力量，同时有效地降低了出现有害的方言碎片的可能性。

你可能就是单纯地欣赏Lisp这门编程语言本身。那从这点说，可以把Clojure当作是一种新的Lisp来了解。从这个层面上看，Clojure是成功的。

2. 并发支持

Clojure的并发之路有可能彻底改变了我们设计并发系统的方式。STM由于其新颖性确实可能会给开发者带来一定的负担，但这是头一回，语言通过检测状态改变是否发生在（受适当保护的）函数内来保护开发者。如果不在事务里，就不能修改状态。

① http://www.tbray.org/ongoing/When/200x/2009/12/01/Clojure-Theses。——原书注

3. Java集成

Clojure有非常好的Java集成。它透明地使用了诸如字符串和数字等一些本地类型。Clojure的亮点在于支持和JVM紧密集成，因此使得Clojure类型可以完全参与到Java应用中。很快你就会明白Clojure自身的很多部分都是在JVM上实现的。

4. 延迟计算

Clojure增加了强大的延迟计算特性。延迟计算可以帮助简化问题。你只是简单地体验了一把延迟序列影响解决问题的方式。通过将计算推迟到实际需要时才执行，或者干脆避免执行，延迟序列可以显著减少计算开销。最后，延迟问题提供了解决困难问题的又一项工具。可以用延迟序列代替递归、迭代或者已实现的集合。

5. 数据即代码

程序就是列表。像其他任何Lisp一样，你可以把数据当作代码。使用Ruby时，我注意到了用程序写程序的价值。我认为对任何编程语言来说，这都是最重要的能力。函数式程序通过高阶函数提供元编程。Lisp将这个想法进一步扩展，把数据当作代码求值。

7.5.3 不足之处

Clojure是一门坚定以通用编程语言为目标的语言。它是否真的能在JVM上获得广泛成功还有待检验。Clojure有很多美妙的抽象，但要真正接受并安全有效地使用这些特性，需要程序员具备很高的教育水平和天赋。以下列举出我认为的一些不足之处。

1. 前缀表达法

将代码表示为列表形式是Lisp最强大的特性之一，但也有代价，即前缀表示法[①]。典型的面向对象语言语法与之差异极大。要调整适应前缀表示法并不容易。它需要更好的记忆力，并且要求开发者由内向外地理解代码，而不是由外向内。有时，我觉得阅读Clojure代码迫使我过早地去了解过多细节。好处是，Lisp语法锻炼了我的短期记忆。身处难关志在远，万千峻险终有尽。

2. 可读性

数据及代码的另一个成本就是多得令人压抑的括号。为人和为计算机优化完全不是一回事。括号的位置和数量仍然是一个问题。Lisp开发者高度依赖编辑器提供的括号匹配反馈，但是工具永远不能完全掩盖可读性问题。感谢Rich对这个问题所作出的改进，但这仍是一个问题。

3. 学习曲线

Clojure内容丰富，但是其学习曲线让人感到沮丧。你需要一队极有天赋和经验的人马才能用Lisp开展工作。延迟序列、函数式编程、宏扩展，事务内存以及精密复杂的方法，所有这些都是需要时间才能掌握的强大概念。

4. 受限的Lisp

所有的妥协都有其代价。由于在JVM上，Clojure限制了尾递归优化，Clojure程序员必须用可怕的`recur`语法。不信就试试看分别用递归和`loop/recur`来实现返回序列`x`长度的函数`(size x)`。

① Clojure确实有从左向右宏，->>和->，它们能稍微缓解一些问题。

消灭用户定义的宏读取器也是一个典型的例子。好处很明显，宏读取器被滥用时，可以导致语言分裂。代价也很明显，你又失去一样元编程工具。

5. 亲和度

Ruby以及早期Java最美的方面之一就是它们作为编程语言的亲和度，这两种语言都相对简单易学。Clojure则对开发者提出了极大的要求，它包含了太多的抽象工具和概念，有时让人不堪重负。

7.5.4　最后思考

Clojure的优势和缺点大部分和它的力量与灵活性有关。的确，你可能需要非常努力才能学会Clojure。事实上，如果你是一名Java开发者，那你已经很努力了。你已经花掉了自己的时间去掌握Java应用层的各种抽象。你希望通过Spring或者面向方面编程获得松耦合。你只是没能从语言层面附带的灵活性中充分受益。对很多人来说，这种权衡起过作用。原谅我大胆地推测，来自并发和复杂性的新需求会继续使Java平台越来越难以为继。

如果你需要一种极端的编程模型并且愿意付出学习语言的代价，Clojure非常合适。我认为，如果你的团队成员受过严格的训练，并且希望扩大影响，那么Clojure是一种很有利用价值的语言。你可以用Clojure更快地创建出更好的软件。

第 8 章 Haskell

逻辑是草地上几只吱吱作声的小鸟在鸣叫。

——Spock

对于很多函数式编程的忠实拥趸来说，Haskell 象征着纯洁和自由。它的功能丰富且强大，但拥有这些功能是需要付出一定代价的。你不可能轻易地就掌握这门语言，因为Haskell 会迫使你去了解关于函数式编程的全部内容。想想《星际迷航》的Spock吧，他上面说的那句话①很有代表性，完美地结合了逻辑和真理。他性格中拥有的那种坚定的纯洁性，这使他得到了几代人的爱戴。当Scala、Erlang和Clojure还允许你少量使用命令式编程概念的时候，Haskell却没有留下任何的回旋余地。在使用Haskell做I/O操作或状态累积（accumulate state）时，你将遇到这门纯函数语言所带来的挑战。

8.1 Haskell 简介

和以往一样，如果想了解一门语言为何包含那些妥协方案，就应该从它的历史开始。在20世纪80年代中前期，纯函数编程领域涌现出了多门语言。纯函数式编程和我们曾在Clojure语言中见到过的惰性处理（lazy processing）等关键概念引领着新研究的方向。1987 年的“函数式编程语言与计算机体系结构大会”（Functional Programming Languages and Computer Architecture）成立了一个小组，决定建立一个关于纯函数编程语言的开放标准。Haskell就出自于这个小组，它于1990年诞生并于1998年重新修订。目前的标准是Haskell 98，经过多次修订，包括一份Haskell 98 标准的修订版和一个称为Haskell Prime的新版本定义。

因此，Haskell是一门从开始就按照纯函数式编程思想构建的语言，它结合了一些最好的函数式语言思想，并着重于支持惰性处理。

和Scala一样，Haskell也是一门强类型定义的静态类型语言。它的类型模型基于推断理论（inferred）并被公认为是函数语言中最高效的类型系统之一。你会发现该类型系统支持多态语义并有助于人们作出十分整洁清晰的设计。

① *Star Trek:The Original Series* 第41集（I，Mudd）和第42集（The Trouble with Tribbles），导演：Marc Daniels（1967年）。发行商：加利福尼亚州伯班克哥伦比亚广播公司（CBS），派拉蒙电视网（2001年）。（译者注：这部电视系列剧中译名为《星际迷航》，是一部科幻题材的电视剧，描述的是一个乐观的未来世界。Spock是剧中受欢迎的角色之一。他是瓦肯-人类混血儿，因为如此，他不断在严肃的瓦肯逻辑教育与人类感情之间挣扎。）

Haskell也可以支持你在本书中已经看到的一些概念。Haskell支持Erlang风格的模式匹配（pattern matching）和哨兵表达式。你也能在Haskell中发现Clojure风格的惰性求值（lazy evaluation）以及与Clojure和Erlang相同的列表推导语法。

作为一门纯函数式编程语言，Haskell不会产生副作用。然而，一个Haskell函数却可以返回一个有副作用并且会被延迟执行。你将在第三天的学习中看到一个关于这方面内容的例子。在另外一个使用monad概念保存状态的例子中你也可以看到相关内容。

头两天，本书会带你学习一些典型的函数式编程概念，诸如表达式、定义函数、高阶函数，等等。我们也将深入Haskell 的类型模型，你会了解到一些新概念。第三天的学习会让你大开眼界，我们将学习参数化的类型系统和monad等一些比较难以掌握的概念。让我们开始吧。

8.2　第一天：逻辑

正如Spock那样，你会发现Haskell的核心概念很容易掌握。你需要严格地定义函数，对于相同的输入参数，每次你都会得到相同的输出结果。我会使用GHC，即Glasgow Haskell 编译器，6.12.1 版本。它可以运行在多个平台上。你也可以找到并使用其他Haskell 编译器。和以往一样，我们在控制台中输入ghci：

```
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
```

你会看到Haskell 解释器先加载了几个包，然后你就可以输入命令了。

8.2.1　表达式和基本类型

我们将在稍后讨论Haskell的类型系统。在本小节中，我们将集中介绍基本类型的使用。和学习许多其他语言一样，先从数字和一些简单的表达式开始，并尽快地介绍更多的高级类型，比如函数。

1. 数字

现在你知道该怎么做了。输入一些表达式：

```
Prelude> 4
4
Prelude> 4 + 1
5
Prelude> 4 + 1.0
5.0
Prelude> 4 + 2.0 * 5
14.0
```

操作的次序也和你期望的一致。

```
Prelude> 4 * 5 + 1
21
Prelude> 4 * (5 + 1)
24
```

注意，你可以使用括号将多个操作组合在一起。你已经看到了几个数字类型了。让我们再来学习一些字符数据。

2. 字符数据

字符串用双引号表示，就像下面这样：

```
Prelude> "hello"
"hello"
Prelude> "hello" + " world"

<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of `+' at <interactive>:1:0-17
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: "hello" + " world"
    In the definition of `it': it = "hello" + " world"
Prelude> "hello" ++ " world"
"hello world"
```

注意，连接两个字符串使用++操作符而不是+操作符。单个字符可以这样表示：

```
Prelude> 'a'
'a'
Prelude> ['a', 'b']
"ab"
```

注意，字符串只是一个字符列表。我们再来简单学习一些布尔值吧。

3. 布尔类型

布尔类型是另外一种基本类型，它的使用方式与本书中其他中缀记号语言的布尔类型一致。下面的相等表达式和不等表达式都返回布尔值：

```
Prelude> (4 + 5) == 9
True
Prelude> (5 + 5) /= 10
False
```

尝试一个if/then语句：

```
Prelude> if (5 == 5) then "true"

<interactive>:1:23: parse error (possibly incorrect indentation)
```

这是Haskell与本书其他语言的第一个主要不同点。在Haskell中，缩进是有特殊意义的。Haskell可以猜出有一个后续行缩进不正确。我们稍后会看到一些缩进结构，不过我们不会讨论用于控制缩进模式的布局。只需要模仿你在这里所看到的缩进方法，就不会遇到什么阻碍。让我们看一个完整地使用了if/then/else的语句吧：

```
Prelude> if (5 == 5) then "true" else "false"
"true"
```

8

在Haskell中，if是一个函数，不是一个控制结构。这意味着它和其他函数一样有返回值。试试给if传一些true/false值：

```
Prelude> if 1 then "true" else "false"

<interactive>:1:3:
    No instance for (Num Bool)
      arising from the literal `1' at <interactive>:1:3
    ...
```

Haskell是强类型的。if只严格地接受布尔类型参数。接下来，我们让控制台产生另一种类型冲突：

```
Prelude> "one" + 1

<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of `+' at <interactive>:1:0-8
      ...
```

这个错误消息让我们第一次窥视到Haskell的类型系统。它告诉我们"没有接受一个数字Num和一个字符串列表[Char]作为参数且名为+的函数"。注意，我们并没有告知Haskell这些参数是什么类型。这门语言可通过上下文的线索推断出类型。任何时候，你都可以看到Haskell的类型推断在做什么。你可以使用:t，也可以打开:t选项达到同样的目的，像下面这样：

```
Prelude> :set +t
Prelude> 5
5
it :: Integer
Prelude> 5.0
5.0
it :: Double
Prelude> "hello"
"hello"
it :: [Char]
Prelude> (5 == (2 + 3))
True
it :: Bool
```

现在，在每个表达式后，你都可以看到该表达式返回的类型。不过我需要提醒你，将:t用于数字会产生令人困惑的结果，这与数字和控制台之间的相互作用有关。尝试用一下:t函数：

```
Prelude> :t 5
5 :: (Num t) => t
```

这与之前得到的类型it :: Integer不同。控制台会采取更泛化的方式看待数字。除非你打开了:t 选项。否则你将得到一个类，而不是一个单纯的类型。类是用于描述一系列相似类型的。我们将在8.4 小节中深入学习类。

8.2.2　函数

函数是整个Haskell编程范型的核心。由于Haskell既是强类型语言，又是静态语言，因此每个

函数的定义都包含两个部分：一个可选的类型规格说明和一份函数的具体实现。我们会快速学习那些你已经在其他语言中见到过的概念，请跟紧点儿。

1. 定义基本函数

Haskell函数按惯例通常包含两个部分：类型声明和函数声明。

开始先在控制台里定义函数。使用`let`函数将值与实现绑定。在定义一个函数前，试一下`let`。和Lisp一样，在Haskell中`let`用于在局部作用域内将变量与函数绑定。

```
Prelude> let x = 10
Prelude> x
10
```

当你编写Haskell模块时，你可以像下面这样声明一个函数：

```
double x = x * 2
```

然而在控制台里，我们在局部作用域内使用`let`定义一个函数。定义后就可以使用它了。下面是一个简单的`double`函数的例子：

```
Prelude> let double x = x * 2
Prelude> double 2
4
```

现在，我们换成在文件中编码，这样就可以使用多行定义了。使用GHC，完整的`double`函数的定义如下：

haskell/double.hs

```
module Main where

    double x = x + x
```

注意，我们增加了一个名为`Main`的模块。在Haskell中，模块用于将相关代码放到一个相同的作用域里。`Main`模块很特殊，它是顶级的模块。现在我们把注意力集中在`double`函数上。在控制台中加载`Main`模块并像下面这样使用它：

```
Prelude> :load double.hs
[1 of 1] Compiling Main             ( double.hs, interpreted )
Ok, modules loaded: Main.
*Main> double 5
10
```

到目前为止，我们还没有显式定义一个类型。但Haskell会为我们推断出每个类型。每个函数背后都会有一个隐含的类型定义。下面的示例是一个包含了类型定义的函数：

haskell/double_with_type.hs

```
module Main where

    double :: Integer -> Integer
    double x = x + x
```

可以像之前那样加载和使用它：

```
[1 of 1] Compiling Main             ( double_with_type.hs, interpreted )
Ok, modules loaded: Main.
*Main> double 5
10
```

你可以看到这个新函数的类型：

```
*Main> :t double
double :: Integer -> Integer
```

这个定义的含义是函数double 接受一个Integer 类型参数（第一个Integer）并返回一个Integer 类型返回值。

这个类型定义是有局限的。如果你回到前面，重温一下那个无类型版本的double 函数定义，你就会发现一些其他内容：

```
*Main> :t double
double :: (Num a) => a -> a
```

现在，类型定义已经完全不同。在这个定义中，a是一个类型变量。这个定义的含义是“函数double 接受一个具有类型a 的变量作为参数，并且返回一个具有同样类型a的结果”。有了这样一个增强型定义，我们就可以将该函数用于任何支持+函数的类型了。让我们开启这个强大的功能吧。我们来看一个关于阶乘的更有趣的实现吧。

2. 递归

先从一个短小的递归开始。下面是一个在控制台里用一行递归实现的阶乘：

```
Prelude> let fact x = if x == 0 then 1 else fact (x - 1) * x
Prelude> fact 3
6
```

这是一个开始。如果x是0，那么x的阶乘结果将是1，否则结果为fact(x - 1) * x。我们可以使用模式匹配来编写出一个更好的阶乘实现。事实上，模式匹配的语法无论是形式上还是行为上都与Erlang的模式匹配如出一辙：

haskell/factorial.hs

```
module Main where
    factorial :: Integer -> Integer
    factorial 0 = 1
    factorial x = x * factorial (x - 1)
```

这个定义有三行代码。第一行声明了参数和返回值的类型。后两行是两个不同的函数式定义，采用哪个定义取决于对输入参数进行模式匹配的结果。0的阶乘是1，而n的阶乘是factorial x = x * factorial (x - 1)。这个定义看起来很像数学定义。在这个例子中，模式的排列顺序是至关重要的。Haskell会使用第一次匹配成功的结果。如果你想改变模式排列的顺序，你可以使用门卫表达式。在Haskell中，哨兵表达式是约束参数值的条件，就像下面这样：

haskell/fact_with_guard.hs

```
module Main where
    factorial :: Integer -> Integer
```

```
factorial x
    | x > 1 = x * factorial (x - 1)
    | otherwise = 1
```

在这个例子中，哨兵表达式的左边是布尔值，右边是待应用于参数的函数实现。当哨兵条件得到满足时，Haskell就会调用相应的函数。哨兵表达式经常用来替代模式匹配，这里用它来初始化递归的基本条件。

8.2.3 元组和列表

正如你在其他语言中所看到的那样，Haskell依靠尾递归优化高效地处理递归。接下来看看用Haskell实现的几个不同版本的斐波那契序列。首先，我们来看一个简单的例子：

haskell/fib.hs

```
module Main where
    fib :: Integer -> Integer
    fib 0 = 1
    fib 1 = 1
    fib x = fib (x - 1) + fib (x - 2)
```

这很简单。`fib 0`或`fib 1`都是1，`fib x`是`fib (x - 1) + fib (x - 2)`。不过这个解决方法不够高效。下面来构建一个更为高效的解决方法吧。

1. 使用元组编程

我们可以使用元组提供一个更为高效的实现。元组是拥有固定数量元素的集合。Haskell的元组由括号内以逗号分隔的元素组成。这个实现使用一系列连续的斐波那契数构建元组，并且使用计数器辅助进行递归操作。下面是这个基本的解决方法：

```
fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
fibTuple (x, y, 0) = (x, y, 0)
fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)
```

`fibTuple`接受一个三元组作为参数，并返回一个三元组。这里要注意，用一个三元组作为一个参数与接受三个参数是不同的。要使用这个函数，用两个数字0和1开始递归。我们还提供了一个计数器。随着计数器的倒数，通过前两个数字获得后续序列中较大的数字。`fibTuple (0, 1 ,4)`的连续调用如下所示：

- fibTuple (0, 1, 4)
- fibTuple (1, 1, 3)
- fibTuple (1, 2, 2)
- fibTuple (2, 3, 1)
- fibTuple (3, 5, 0)

你可以像下面这样运行这个程序：

```
Prelude> :load fib_tuple.hs
[1 of 1] Compiling Main             ( fib_tuple.hs, interpreted )
Ok, modules loaded: Main.
*Main> fibTuple(0, 1, 4)
(3, 5, 0)
```

答案在返回结果的第一个元素位置上。我们可以用下面的方式获得答案：

```
fibResult :: (Integer, Integer, Integer) -> Integer
fibResult (x, y, z) = x
```

我们刚刚使用模式匹配获得了第一个位置上的元素。我们可以简化这个使用模型，如下所示：

```
fib :: Integer -> Integer
fib x = fibResult (fibTuple (0, 1, x))
```

这个函数使用了两个辅助函数构造了一个快速斐波那契序列生成器。下面是该程序的全部代码：

haskell/fib_tuple.hs

```
module Main where
    fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
    fibTuple (x, y, 0) = (x, y, 0)
    fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)

    fibResult :: (Integer, Integer, Integer) -> Integer
    fibResult (x, y, z) = x

    fib :: Integer -> Integer
    fib x = fibResult (fibTuple (0, 1, x))
```

下面是运行结果（立刻得到）：

```
*Main> fib 100
354224848179261915075
*Main> fib 1000
    43466557686937456435688527675040625802564660517371780
    40248172908953655541794905189040387984007925516929592
    25930803226347752096896232398733224711616429964409065
    33187938298969649928516003704476137795166849228875
```

让我们尝试另外一种使用函数组合的方法。

2. 使用元组和组合

有时，你需要将函数串联地组合在一起，并将结果从一个函数传给另一个。下面例子中我们通过匹配tail的head来获得列表的第二个元素：

```
*Main> let second = head . tail
*Main> second [1, 2]
2
*Main> second [3, 4, 5]
4
```

我们刚刚在控制台中定义了一个函数。second = head.tail等价于second lst = head (tail lst)。我们将第一个函数的返回结果传给另外一个函数。利用这个特性实现了另外一个

斐波那契序列。

和以前一样，这次使用一个二元组，但没有使用计数器：

```
fibNextPair :: (Integer, Integer) -> (Integer, Integer)
fibNextPair (x, y) = (y, x + y)
```

已知序列中的两个数字，就可以计算出下一个。下面要做的就是递归地计算出序列中的下一个值：

```
fibNthPair :: Integer -> (Integer, Integer)
fibNthPair 1 = (1, 1)
fibNthPair n = fibNextPair (fibNthPair (n - 1))
```

基本情况是当n为1时值为(1, 1)。由此，仅需根据上一个值计算出序列的下一个元素。这样可以得到序列中由任意两个数字组成的二元组：

```
*Main> fibNthPair(8)
(21,34)
*Main> fibNthPair(9)
(34,55)
*Main> fibNthPair(10)
(55,89)
```

现在，剩下要做的只是匹配每个二元组的第一个元素，并将它们结合在一起放入一个序列中。我们将使用一个由fst和fibNthPair组成的简便的函数组合，使用fst抓取第一个元素，使用fibNthPair构建一个二元组：

haskell/fib_pair.hs

```
module Main where
    fibNextPair :: (Integer, Integer) -> (Integer, Integer)
    fibNextPair (x, y) = (y, x + y)

    fibNthPair :: Integer -> (Integer, Integer)
    fibNthPair 1 = (1, 1)
    fibNthPair n = fibNextPair (fibNthPair (n - 1))

    fib :: Integer -> Integer
    fib = fst . fibNthPair
```

换句话说，上述代码就是用来获取第n个元组的第一个元素。到此任务完成。在用元组完成一些任务后，下面我们再用列表解决一些问题吧。

3. 遍历列表

你在很多语言中看到过列表。我不会一味地老调重弹，不过我会重温一个基本的递归例子并介绍一些你之前从未见到过的函数。在任何一个绑定操作过程中都可以将列表拆分为head和tail两部分，就像一条let语句或一个模式匹配：

```
let (h:t) = [1, 2, 3, 4]
*Main> h
1
*Main> t
[2,3,4]
```

我们将列表[1, 2, 3, 4]绑定到(h:t)。可以把这个构造看成是你曾经在Prolog、Erlang和Scala中所见过的head|tail结构。用这种方式，可以进行一些简单的递归定义。下面是一个列表的size和prod函数：

haskell/lists.hs

```
module Main where
    size [] = 0
    size (h:t) = 1 + size t

    prod [] = 1
    prod (h:t) = h * prod t
```

我将使用Haskell的类型推断处理这些函数的类型，不过意图是清晰的。列表的size就是1加上列表tail部分的size。

```
Prelude> :load lists.hs
[1 of 1] Compiling Main
    ( lists.hs, interpreted )
Ok, modules loaded: Main.
*Main> size "Fascinating."
12
```

zip是一个用于合并列表的强大工具，下面是这个函数的一个实例：

```
*Main> zip "kirk" "spock"
[("kirk","spock")]
```

我们用两个元素构造了一个元组。你也可以将两个列表合并在一起，像下面这样：

```
Prelude> zip ["kirk", "spock"] ["enterprise", "reliant"]
[("kirk","enterprise"),("spock","reliant")]
```

这是一个合并两个列表的有效方法。

到目前为止，你在Haskell中看到的特性与其他语言拥有的特性非常相似。现在，我们将开始学习使用一些更为高级的构造结构。你将看到一些高级列表，包括范围（range）和列表推导（list comprehension）。

8.2.4　生成列表

我们已经看过一些使用递归处理列表的方法了。在本节中，我们将介绍一些用于生成新列表的方法。主要是介绍递归、范围以及列表推导。

1. 递归

用于构建列表的最基本的结构单元是:操作符，它将head和tail两部分合并，形成一个新列表。你曾经看到过这个操作符在调用一个递归函数时被逆向用于模式匹配过程中。

```
Prelude> let h:t = [1, 2, 3]
Prelude> h
1
Prelude> t
[2,3]
```

我们也可以将`:`用于构建，而不是解构。

下面是一个用`:`构建列表的例子：

```
Prelude> 1:[2, 3]
[1,2,3]
```

记住，列表是同构的（homogeneous）。你不能将一个列表类型元素合并到一个数字列表中，例如：

```
Prelude> [1]:[2, 3]

<interactive>:1:8:
    No instance for (Num [t])
      arising from the literal '3' at <interactive>:1:8
```

不过，你可以将一个列表类型元素合并到一个列表的列表中，或是一个空列表中：

```
Prelude> [1]:[[2], [3, 4]]
[[1],[2],[3,4]]
Prelude> [1]:[]
[[1]]
```

下面是列表构建的一个实例。我们想要创建一个函数，该函数返回一个由列表中所有偶数构成的列表。实现这个函数的一种方式是使用列表构建：

haskell/all_even.hs

```
module Main where
    allEven :: [Integer] -> [Integer]
    allEven [] = []
    allEven (h:t) = if even h then h:allEven t else allEven t
```

函数接受一个整数列表作为参数，并返回由偶数组成的列表。对空列表调用`allEven`将返回空列表。如果存在这样的一个列表，它的`head`是一个偶数，则将这个`head`与对列表`tail`部分执行`allEven`的结果合并在一起。如果`head`是奇数，则丢弃它，并返回对列表`tail`部分执行`allEven`的结果。没问题。接下来看一些其他构建列表的方法。

2. 范围与组合

与Ruby和Scala一样，Haskell拥有作为一等对象的范围（range）和一些支持范围的语法糖。Haskell提供了一种简单的表示范围的形式，即由一个范围的两个端点表示：

```
Prelude> [1..2]
[1,2]
Prelude> [1..4]
[1,2,3,4]
```

指定两个端点，Haskell会计算出这个范围。默认增量为`1`。当Haskell使用默认增量却无法到达端点时会发生什么呢？

```
Prelude> [10..4]
[]
```

你将得到一个空列表。你可以通过指定列表中的下一个元素来指定增量：

```
Prelude> [10, 8 .. 4]
[10,8,6,4]
```

你也可以使用分数：

```
Prelude> [10, 9.5 .. 4]
[10.0,9.5,9.0,8.5,8.0,7.5,7.0,6.5,6.0,5.5,5.0,4.5,4.0]
```

范围是一种创建序列的语法糖。序列不需要边界。和Clojure一样，你可以获取一个序列中的一些元素：

```
Prelude> take 5 [ 1 ..]
[1,2,3,4,5]
Prelude> take 5 [0, 2 ..]
[0,2,4,6,8]
```

我们将在第二天讨论更多有关惰性序列（lazy sequence）的内容。现在，看看另外一种自动生成列表的方法，列表推导（list comprehension）。

3. 列表推导

我们第一次看到列表推导是在Erlang那一章中。在Haskell中，列表推导的工作方式与Erlang相同。在左侧，你会看到一个表达式。在右侧，你会看到生成器和过滤器，就像你在Erlang中看到的那样。我们来看一些例子。要将一个列表中的所有元素值加倍，可以这么做：

```
Prelude> [x * 2 | x <- [1, 2, 3]]
[2,4,6]
```

这个列表推导的含义是“采集x * 2，其中x来自于列表[1, 2, 3]”。

和Erlang一样，我们可以在列表推导中使用模式匹配。假设用一个由一些点组成的列表表示一个多边形，并且按对角线翻转这个多边形，只需要将x和y调换位置即可，像下面这样：

```
Prelude> [ (y, x) | (x, y) <- [(1, 2), (2, 3), (3, 1)]]
[(2,1),(3,2),(1,3)]
```

或者，要水平翻转多边形，可以用4减去x，像下面这样：

```
Prelude> [ (4 - x, y) | (x, y) <- [(1, 2), (2, 3), (3, 1)]]
[(3,2),(2,3),(1,1)]
```

我们还可以计算出所有可能的组合。要找出所有可能的由两人组成的先头部队，而且这两个人要来自于Kirk、Spock或McCoy这组人当中，实现代码如下所示：

```
Prelude> let crew = ["Kirk", "Spock", "McCoy"]
Prelude> [(a, b) | a <- crew, b <- crew]
[("Kirk","Kirk"),("Kirk","Spock"),("Kirk","McCoy"),
("Spock","Kirk"),("Spock","Spock"),("Spock","McCoy"),
("McCoy","Kirk"),("McCoy","Spock"),("McCoy","McCoy")]
```

这个组合几乎可以工作，不过这里没有剔除两个元素重复的情况。我们可以为列表推导增加一个过滤条件，像下面这样：

```
Prelude> [(a, b) | a <- crew, b <- crew, a /= b]
[("Kirk","Spock"),("Kirk","McCoy"),("Spock","Kirk"),
("Spock","McCoy"),("McCoy","Kirk"),("McCoy","Spock")]
```

这个稍微好一些，不过没有关注元组中元素的次序。我们可以做得更好一点，只包含元素按顺序排列的结果，滤掉其他结果：

```
Prelude> [(a, b) | a <- crew, b <- crew, a < b]
[("Kirk","Spock"),("Kirk","McCoy"),("McCoy","Spock")]
```

使用一个短小简单的列表推导，我们得到了答案。列表推导是一个用于快速构建和转换列表的重要工具。

8.2.5 Philip Wadler访谈录

现在，你已经看到了一些Haskell的核心特性，让我们来看看一位来自Haskell设计委员会的重要成员想说些什么。Philip Wadler是就职于爱丁堡大学的一名从事理论计算机科学研究的教授，他不仅是一位Haskell的积极贡献者，而且还是Java和XQuery的贡献者。此前，他先后工作或就读于Avaya实验室、贝尔实验室、格拉斯哥大学（Glasgow）、查尔摩斯大学（Chalmers）、牛津大学、卡耐基梅隆大学（CMU）、施乐帕洛阿尔托研究中心以及斯坦福大学。

Bruce：你的团队为什么要开发Haskell？

Philip：在20世纪80年代后期，有大量不同的团体进行了函数式编程语言的设计和实现。我们意识到大家一起合作的力量将比各自为战更强大。我们最初的目标有些不那么谦逊：我们想要这门语言成为研究的基础，适于教学，并能够胜任工业用途。在一篇名为“History of Programming Languages”（程序设计语言的历史）的论文中，我们详尽记录了这门语言的完整的历史。

Bruce：关于这门语言你最喜欢它哪一点呢？

Philip：我真的很喜欢使用列表推导进行编程。非常高兴地看到它们最终走进了其他语言，像Python。

类型类（type class）提供了一种简单的泛型编程的形式。你定义一个数据类型，并且只需加上一个`derived`关键字，就可以得到诸如值比较、与字符串相互转换等例程。我发现这种形式非常方便，我在使用其他语言时也很想念这种形式。

任何一门优秀的编程语言实际上都会变成一种扩展自己的手段，它通过嵌入其他适合特定任务的语言的方式扩展自己。Haskell尤其适合作为一个嵌入其他语言的工具。惰性机制、lambda表达式、monad和箭头记号、类型类、表达性优异的类型系统以及模板，Haskell支持用各种不同的方式实现扩展。

Bruce：如果能让时光倒流，你想改变哪些特性？

Philip：随着分布式结构变得越来越重要，我们需要关注运行在多个机器上的程序，这些程序将数值从一个机器发到另外一个机器。当发送一个数值时，你可能更想要的是这个值本身[急切求值（eager evaluation）]，而不是一个通过求值才能产生这个值的程序（以及这个程序的所有自由变量的值）。所以，在分布式世界中，我认为默认采用渴望求值的方式更好，不过当你需要的时候使用惰性方法也会很容易。

Bruce：你见到过的用Haskell解决的最有意思的问题是什么？

Philip：我一直在为Haskell寻找用途。记得多年前，我吃惊地看到Haskell被用在自然语言处

8

理中。并且自那时起多年后Haskell又被用于一个抗击艾滋病的蛋白质折叠的应用中。我刚刚看了一下Haskell社区的主页,上面列出了40个Haskell的工业应用。现在Haskell在金融业也有许多用户:荷兰银行、瑞士信贷、德国银行以及英国渣打银行。Facebook采用Haskell实现了一个内部使用的更新PHP代码的工具。我最喜欢的是一个使用Haskell实现的垃圾回收应用——不是在软件中使用的那种内存垃圾回收，而是真正的生活垃圾回收……是用在垃圾车上的编程引擎。

8.2.6　第一天我们学到了什么

Haskell是一门函数式编程语言。它第一个显著的特性就是它是一门纯函数式语言，对一个函数使用相同参数，总是可以得到相同的结果。它没有副作用。我们在第一天花了大部分时间了解语言的特性，你曾在本书其他语言中见到过这些特性。

我们首先学习了基本表达式和简单数据类型。由于没有可变的变量赋值机制，所以使用递归定义了一些简单数学函数，并用递归处理列表。我们使用基本的Haskell表达式，并将它们结合在一起形成新函数。我们看到了类似Erlang和Scala中的模式匹配和哨兵机制。和你在Erlang中看到的一样，使用了列表和元组作为基本的集合。

最后，我们学习了构建列表，并借此了解到列表推导、范围以及惰性序列。让我们将这里的一些想法应用到实际中吧。

8.2.7　第一天自习

到现在，如果你已经完成了本书所有其他函数式编程语言的学习，那么编写函数式程序将变得更加容易。在本节中，我将给你加一些难度。

找

- Haskell维基。
- 提供编译器选择支持的在线Haskell团体。

做

- 你能找到多少种实现`allEven`的方法?
- 编写一个函数，它以一个列表作为参数并返回逆序后的列表。
- 从五个颜色黑、白、蓝、黄和红任选出两个组合在一起，编写一个函数，计算出所有可能的组合。注意，你只能包含`(black, blue)`和`(blue, black)`两者中的一个。
- 编写一个列表推导来构建一个儿童乘法表。这个表应该是一个由三元组组成的列表，三元组的前两个整数元素的取值范围是1~12，第三个元素为前两个元素的乘积。
- 使用Haskell解决地图着色问题（请参考4.2节）。

8.3　第二天：Spock 的超凡力量

与一些人一起工作,你可能在很长的一段时间里都不会注意到他们的优秀品质。不过与Spock一起工作，你会很容易看到他超凡的力量，他英明果断，很有逻辑性，并且可完全预知事情的发

展。Haskell的超凡优势也恰是逻辑的预知能力和简单性。许多大学在程序推理课程中教授Haskell。Haskell在进行正确性证明上比命令式语言容易得多。在本节中，我们将深入研究这个实用的、可以提供更好的预知能力的概念。我们从高阶（higher-order）函数开始，然后讨论Haskell组合高阶函数的策略。这里将会介绍偏应用函数和柯里化。我们最后会看看惰性计算（lazy computation）。这势必会是充实的一天。让我们开始吧。

8.3.1 高阶函数

这本书中的每门语言都涵盖了高阶编程的思想。Haskell更广泛地依赖这个概念。我们将快速学习匿名函数，并且在很多预置的操作列表的函数中使用它们。由于你已经看到过这些概念，并且有了不错的基础，所以我会用比其他语言更快的速度完成这门语言的学习。首先从匿名函数开始。

1. 匿名函数

你可能已经预料到，Haskell中的匿名函数语法非常简单。其语法格式是`(\param1.. paramn -> function_body)`。像下面这样试一下：

```
Prelude> (\x -> x) "Logical."
"Logical."
Prelude> (\x -> x ++ " captain.") "Logical,"
"Logical, captain."
```

单独使用匿名函数时，它们并没有增加什么功能。不过与其它函数结合在一起使用，它们的功能将变得非常强大。

2. map和where

首先，定义了一个匿名函数，它只是返回第一个参数。接下来，定义了一个用于附加字符串的函数。和你在其他语言中已经看到的一样，匿名函数对于列表库来说是一个非常重要的特性。Haskell内置了一个map函数：

```
map (\x -> x * x) [1, 2, 3]
```

将`map`函数应用于一个匿名函数和一个列表上。`map`会将这个匿名函数应用到列表中的每一项并收集返回结果。这里没有什么令人惊诧的东西，不过这种形式可能有些让人陌生而难以一次全部理解消化。我们可以将所有这些都打包成一个函数，并用局部作用域函数替代匿名函数，如下所示：

haskell/map.hs

```
module Main where
    squareAll list = map square list
        where square x = x * x
```

我们已经定义了一个名为`squareAll`的函数，它接受一个名为list的参数。接下来，使用`map`将一个名为`square`的函数应用到列表的所有元素上。然后，使用一个称为`where`的新特性声明`square`函数的一个的局部作用域的版本。你不必为`where`绑定函数，但可以绑定任何变量。本章

8

后续还会有一些有关where的例子，下面是运行结果：

```
*Main> :load map.hs
[1 of 1] Compiling Main             ( map.hs, interpreted )
Ok, modules loaded: Main.
*Main> squareAll [1, 2, 3]
[1,4,9]
```

你也可以将map和函数的某一部分一起使用，这称为section，像下面这样：

```
Prelude> map (+ 1) [1, 2, 3]
[2,3,4]
```

(+1)实际上是一个偏应用函数。函数+接受两个参数，但实际上只提供了一个。最终结果是得到了一个类似(x + 1)且仅接受一个参数x的函数。

3. filter、foldl和foldr

下一个常用的函数是filter，这个函数会对一个列表中每个元素进行一个测试，像下面这样：

```
Prelude> odd 5
True
Prelude> filter odd [1, 2, 3, 4, 5]
[1,3,5]
```

你也可以执行fold left和fold right，就像我们在Clojure和Scala中使用的那样。你即将使用的函数是foldl和foldr的变种：

```
Prelude> foldl (\x carryOver -> carryOver + x) 0 [1 .. 10]
55
```

我们使用一个初值为0的进位（carryover），然后将这个函数应用于列表中的每个元素，使用函数返回结果作为参数carryOver，使用列表中的每个元素作为另外一个参数。当你用操作符进行fold操作时，另外一种fold的形式更加简便：

```
Prelude> foldl1 (+) [1 .. 3]
6
```

这里使用操作符+作为一个接受两个参数并返回一个整数的纯函数。其结果和下面计算的结果相同：

```
Prelude> 1 + 2 + 3
6
```

你也可以使用foldr1从右到左地进行fold操作。

你可能已经想到，Haskell在标准库中提供了许多其他的列表操作函数，这些函数多数都使用到了高阶函数。与其花费一章的时间讲解它们，不如让你自己发掘它们。现在，我们将深入学习Haskell将函数组合在一起工作的方法。

8.3.2　偏应用函数和柯里化

我们已经对函数组合与偏应用函数做了简单介绍。这些概念非常重要并且是Haskell的核心概念。我们应该花更多时间在这里。

每个Haskell函数都只有一个参数。你也许会问自己："如果这是真的，那么如何编写一个像+这样的将两个数字相加在一起的函数呢？"

事实上，这的确是真的。每个函数的确只拥有一个参数。为了简化类型的语法，我们来定义一个名为prod的函数：

```
Prelude> let prod x y = x * y
Prelude> prod 3 4
12
```

我们定义了一个函数，它可以工作。我们来看一下这个函数的类型：

```
Prelude> :t prod
prod :: (Num a) => a -> a -> a
```

Num a =>这部分的含义是："在接下来的类型定义中，a是一个Num类型变量。"之前你已经看见过其余的定义了。当时我为了简化一些事情而在这个含义上面说了谎。现在，是澄清事实的时候了。Haskell使用了一个概念，将拥有多个参数的函数拆分为多个只有一个参数的函数。Haskell使用偏应用来完成这个工作。

不要被名词术语所困扰。偏应用绑定某些参数，不过不是所有的。例如，我们可以对prod运用偏应用来创建一些其他函数：

```
Prelude> let double = prod 2
Prelude> let triple = prod 3
```

先来看看这些函数左侧的定义。之前定义的prod有两个参数，不过这里仅使用了第一个参数。这样一来，prod 2的计算变得容易，只是将最初函数版本prod x y = x * y中的x替换成2，得到prod y = 2 * y。这个函数工作起来与你预期的相同：

```
Prelude> double 3
6
Prelude> triple 4
12
```

谜底揭晓。当Haskell计算prod 2 4时，它实际上计算(prod 2) 4，像下面这样。

- 首先，应用prod 2。这将返回函数(\y -> 2 * y)。
- 然后，应用(\y -> 2 * y) 4或2 * 4，结果为8。

这个过程称为柯里化，并且几乎每个Haskell中的多参数函数都是柯里化的。这样Haskell就具有更大的灵活性以及更简单的语法。大多数情况下，你实际上并不需要考虑它，因为柯里化和未柯里化的函数的结果是相同的。

8.3.3 惰性求值

和Clojure的序列库相似，Haskell广泛使用了惰性求值。有了惰性求值，你可以构建返回无穷列表的函数。通常，你会使用列表构建方法来产生一个无穷列表。看看下面这个例子，它构建了一个无穷范围，从x开始，步长为y：

haskell/my_range.hs

```
module Main where
    myRange start step = start:(myRange (start + step) step)
```

这个函数的语法有些奇特，不过整体的效果很美妙。我们构建了一个名为myRange的函数，它接受一个范围的起点和步长作为参数。我们使用列表组合来构建这个列表，用start参数作为列表的head，用(myRange (start + step) step)作为列表的tail部分。下面是myRange 1 1的后续求值操作过程：

- 1:myRange (2 1)
- 1:2:myRange (3 1)
- 1:2:3:myRange (4 1)

……等等。

这个递归将无休止地进行下去，所以一般将这个函数与其他可以限制递归过程的函数一起使用。首先确保已经加载了my_range.hs：

```
*Main> take 10 (myRange 10 1)
[10,11,12,13,14,15,16,17,18,19]
*Main> take 5 (myRange 0 5)
[0,5,10,15,20]
```

列表构造的方法可以使得一些递归函数工作得更加高效。下面是一个斐波那契序列的例子，我们在这个例子中使用了组合惰性求值：

haskell/lazy_fib.hs

```
module Main where
    lazyFib x y = x:(lazyFib y (x + y))

    fib = lazyFib 1 1

    fibNth x = head (drop (x - 1) (take (x) fib))
```

第一个函数构建了一个序列，其中每个数字都是前两个数字的和。我们可以很快地获得一个序列，不过在API上还有改善余地。要想生成一个合适的斐波那契序列，必须用1和1来作为序列的头两个数字，这样fib才能使用前两个数字为lazyFib提供参数。最后，我们拥有不止一个允许用户使用drop和take从序列中抓取一个数字的帮助函数。下面是这些函数应用的实例：

```
*Main> take 5 (lazyFib 0 1)
[1,1,2,3,5]
*Main> take 5 (fib)
[1,1,2,3,5]
*Main> take 5 (drop 20 (lazyFib 0 1))
[10946,17711,28657,46368,75025]
*Main> fibNth 3
2
*Main> fibNth 6
8
```

这三个函数优美且简明。我们定义了一个无穷序列，Haskell只是计算完成这项工作所需的必要部分。当你开始尝试将无穷队列结合在一起时，你就会开始体会到这种乐趣。首先，将两个相差偏移为1的斐波那契序列相加到一起：

```
*Main> take 5 (zipWith (+) fib (drop 1 fib))
[2,3,5,8,13]
```

令人惊奇的是，我们得到了一个斐波那契序列。这些高阶函数在一起工作得很好。函数`zipWith`将两个列表中下标相同的每一项结对，然后把函数+传递给它。我们还可以将范围内的元素翻倍：

```
*Main> take 5 (map (*2) [1 ..])
[2,4,6,8,10]
```

我们使用`map`将偏应用函数`*2`应用于无穷范围`[1..]`，然后从1开始使用这个无穷范围。

函数式编程语言的好处是，你可以用意想不到的方式编写它们。例如，我们可以毫不费劲地使用函数组合将偏应用函数与惰性序列放在一起使用：

```
*Main> take 5 (map ((* 2) . (* 5)) fib)
[10,10,20,30,50]
```

这段代码比较有冲击力，我们把它拆解开来细致分析。从里往外，首先是`(* 5)`。这是一个偏应用函数。传递给该函数的参数都将被乘以5。我们将结果传递给另外一个偏应用函数`(* 2)`。再将这个组合函数传递给`map`，并将这个函数组合应用到无穷`fib`序列的每个元素上。将这个无穷的结果传递给`take 5`，生成一个斐波那契序列的前5个元素，这些元素先被乘以5，后又被乘2。

你可以很容易地看到你是如何构建问题的解决方案的。只是将一个函数传递给下一个函数。在Haskell中，`f . g x`是`f (g x)`的缩写。当按照这种方式构建函数时，你可能想要按从第一个到最后一个的顺序应用这些函数。你可以用“`.`”操作符来达到这个目的。例如，要倒转一个图片，先垂直翻转图片，再水平翻转即可。图片处理器可能会执行类似`(flipHorizontally . flipVertically . invert) image`的代码。

8.3.4　Simon Peyton-Jones访谈录

短暂休息一下。让我们来听听另一位来自Haskell委员会成员的观点吧。Simon Peyton-Jones在伦敦大学学院做了7年讲师，并作为教授供职于格拉斯哥大学9年。在1998年他进入微软研究院（剑桥）之后，他的研究领域主要集中在单处理器和并行主机上实现并应用函数式编程语言。他是本书所用的Haskell编译器的主要设计者。

Bruce：给我讲一些关于创造Haskell的事情吧。

Simon：Haskell很不寻常的一点就是，它是一门成功的由委员会创造的语言。想一下任何一门成功的语言，其最初的实现很可能是由一名开发者或一个非常小的团队完成。Haskell则不同，它最初就是由一个二十几个研究人员组成的国际团体设计的。我们在语言的核心原则上充分地达成共识。Haskell是一门原则性非常强的语言，这样才能保持语言设计的一致性。

此外，在诞生二十多年后，Haskell在受欢迎程度上正迎来一次重要的增长。编程语言通常在

8

其生命周期的头几年要么成功或要么（大多数）失败，而Haskell在这么多年后才取得成功的原因是什么呢？我相信是由于Haskell坚持纯洁性的原则以及不存在副作用。陌生的行为方式阻碍了Haskell成为一门主流编程语言。那些长远的好处会逐渐变得显而易见。无论未来的主流编程语言看起来是否与Haskell相像，我相信它们都会具有强大的用于控制副作用的机制。

Bruce：你最喜欢它哪一点呢？

Simon：除了纯洁性之外，可能Haskell中最不寻常、最具吸引力的特性就是其类型系统了。静态类型是当今现有的使用最为广泛的程序验证技术。成百上千万的程序员每天编写类型（仅是部分规范），并且编译器在每次编译这些程序时都检查这些类型。类型是函数式编程的UML：一种可以关系紧密并形成程序永久组成部分的设计语言。

从第一天的学习起，Haskell的类型系统就拥有不寻常的表现力，主要是因为类型类和更高级别的类型变量。从那时起，Haskell就成了我十分喜欢的用于探索新类型系统想法的实验室。多参数类型类、更高级别类型、一等对象的多态机制、隐式参数、GADT（Generalised algebraic datatype，通用代数数据类型）以及类型体系……我们正乐在其中！更重要的是，我们正在扩展属性的范围以使得它们可以通过类型系统进行静态检查。

Bruce：如果能让时光倒流，你想改变哪些特性？

Simon：我想要一个更好的记录系统（record system）。Haskell的记录系统如此简单是有原因的，但它仍然是一个不足之处。

我想要一个更好的模块系统（module system）。特别是，我想能够发布一个Haskell包P给其他某个人，并说："P需要从其他地方导入接口I和J：你需要提供它们，并且P将提供接口K"。Haskell没有用于此的正式方法。

Bruce：在实际产品中，你见过的最特别的Haskell应用是什么？

Simon：Haskell是一门真正的通用编程语言，这是一个优势，但同时也是一个不足，因为它没有"杀手级应用"（killer app）。也就是说，我们常常可以发现Haskell是一种方法，通过这种方法人们可以想出特别优雅和不寻常的方式来解决问题。看一下Conal Elliot在函数反应式动画（functional reactive animation）方面所做的工作，它让我把一个"时变值"看成一个可以被函数式程序操纵的单一值，通过这种方式它可以改变我的思维。在一个更现实（但很有用）的级别上，Haskell也有许多有关解析器和美化输出组合的库，每个简单接口的后面都封装了强大的智能系统。Jean-Marc Eber向我展示了如何设计一个组合库来描述金融衍生产品，这些东西我自己是从来不会想到的。

在每个例子中，Haskell都支持了一个新的表达层次，从现有的主流语言中获得这些将特别困难。

现在，你有足够的知识使用Haskell去解决一些难度较高的问题了，不过你无法解决一些简单的诸如I/O、状态以及错误处理相关的问题。这些问题促使我们深入学习一些高级理论。在第三天的学习里，我们将学习研究monad。

8.3.5　第二天我们学到了什么

在第二天，我们了解了高阶函数。我们开始使用那些你在本书其他语言中几乎都看到过的相

同类型的列表库。你看到了`map`、几个版本的`fold`以及其他一些诸如`zip`和`zipWith`的函数。我们在固定长度列表上应用了这些函数，接下来采用了一些惰性技术，就像你在Clojure语言中用到的那些。

在完成了高级函数的学习后，我们学会了使用函数并只应用它的一部分参数。这种技术被称为偏应用函数。接下来，使用偏应用函数将一个一次接受多个参数的函数（`f (x,y)`）转换为每次只接受一个参数的函数（`f (x) (y)`）。我们了解到在Haskell中所有函数都可以柯里化，这也解释了为何Haskell函数的类型签名接受多个参数。例如，函数`f x y = x + y`的类型签名为`f :: (Num a) => a -> a -> a`。

我们还学会了函数组合，将函数的返回值作为另一个函数的输入。通过这种方法可以高效地将函数结合在一起。

最后，我们使用了惰性求值。我们可以定义用于构建无穷列表的函数，构建出的无穷列表将根据需要进行处理。通过这种方法可以构建出斐波那契序列，并将函数组合与惰性序列一起使用毫不费力地就构造出一个新的惰性序列。

8.3.6　第二天自习

找

- 可以用来操作列表、字符串或元组的函数。
- 对列表排序的方法。

做

- 编写函数`sort`，接受一个列表作为参数并且返回一个有序的列表。
- 编写函数`sort`，接受一个列表和一个比较两个参数大小的函数作为参数，然后返回一个有序列表。
- 编写一个Haskell函数，将字符串转换为数字。字符串应该以`$2,345,678.99`形式提供，并且可以包含前导零。
- 编写一个函数，该函数接受一个参数`x`，并返回从`x`起始，每两个元素间相隔差值为2的惰性序列。然后，编写另外一个函数，返回从`y`开始，每两个元素间相隔差值为4的惰性序列。通过组合将两个函数合并在一起返回一个从`x+y`开始，每两个元素间相隔差值为7的惰性序列。
- 使用偏应用函数定义两个函数，其中一个将返回一个数值的一半。另外一个函数会将`\n`附加到任意字符串的末尾。

如果你想做一些更有趣的事情，下面有一些难度更高的练习。

- 编写一个函数，用来确定两个整数的最大公约数。
- 创建一个惰性的素数序列。
- 根据恰当的字边界将一个长字符串拆分为多行。
- 为上一个练习添加行号。
- 对上面的练习，将函数加到左侧和右侧，使得每行文字用空格补齐（使得两个页边笔直）。

8

8.4　第三天：心灵融合

在《星际迷航》中，Spock拥有一种特殊的能力，他可以使用一种被他称为心灵融合（mind meld）的能力与某个人建立连接。Haskell爱好者们经常声称他们与Haskell语言间存在这种连接。对许多人来说，对他们影响力最大的语言特性就是类型系统。在长时间使用这门语言后，我可以很容易理解这一切成为事实的缘由。Haskell的类型系统灵活且功能相当强大，它可以推断出我的绝大多数意图，它并不介入我的工作，除非我需要它。当构建函数时，特别是那些由函数组合而成的抽象函数时，我可以得到类型系统提供的完好性检查。

8.4.1　类与类型

Haskell的类型系统是这门语言最强大的特性之一。它支持类型推断，因此可以为程序员减轻许多负担。它也足够健壮，甚至可以捕捉到程序中的一些极细微的错误。它是多态的，这意味着你可以按同样方式对待同一种类型的不同形式。在本节中，我们会看到一些类型相关的例子并构建一些属于我们自己的类型。

1. 基本类型

让我们回顾一下到目前为止已经学过的一些基本类型吧。首先，打开控制台的类型信息选项。

```
Prelude> :set +t
```

现在，我们就可以看到每条语句返回的类型信息了。用一些字符和字符串试试：

```
Prelude> 'c'
'c'
it :: Char
Prelude> "abc"
"abc"
it :: [Char]
Prelude> ['a', 'b', 'c']
"abc"
it :: [Char]
```

控制台总是会返回你最后输入的语句的值，并且你可以将::读作是具有某种类型。对于Haskell来说，字符是一个原生类型。字符串是一个字符的数组。用数组还是用双引号来表示字符数组无关紧要。对于Haskell来说，这两种形式的值都是相同的：

```
Prelude> "abc" == ['a', 'b', 'c']
True
```

下面是一些其他的原生类型：

```
Prelude> True
True
it :: Bool
Prelude> False
False
it :: Bool
```

随着对类型的深入学习，这些想法将帮助我们了解真实发生的一切。接下来定义一些我们自己的类型吧。

2. 用户自定义类型

我们可以通过data关键字定义自己的数据类型。最简单的类型声明使用了一个有限长度的值列表。例如，Boolean类型可以这样定义：

```
data Boolean = True | False
```

这个定义的含义是Boolean类型有单一值，要么是True，要么是False。我们也可以用同样的方式定义自己的类型。考虑下面这盒只有两套花色（suit）和五个级别（rank）的扑克牌吧：

haskell/cards.hs

```
module Main where
    data Suit = Spades | Hearts
    data Rank = Ten | Jack | Queen | King | Ace
```

在这个例子中，Suit和Rand都是类型构造器。我们使用关键字data构造了一个新的用户自定义类型。你可以像这样加载这个模块：

```
*Main> :load cards.hs
[1 of 1] Compiling Main             ( cards.hs, interpreted )
Ok, modules loaded: Main.
*Main> Hearts

<interactive>:1:0:
    No instance for (Show Suit)
      arising from a use of `print' at <interactive>:1:0-5
```

哎呀，发生了什么？Haskell告诉我们控制台尝试显示这些值，但却并不知道如何显示。有一种支持显示自定义类型的快速方法，即当声明用户自定义类型时，需要继承show函数。像下面这样：

haskell/cards-with-show.hs

```
module Main where
    data Suit = Spades | Hearts deriving (Show)
    data Rank = Ten | Jack | Queen | King | Ace deriving (Show)
    type Card = (Rank, Suit)
    type Hand = [Card]
```

注意，我们向系统里加了一些别名。Card是由rank和suit组成的元组，Hand是一个card列表。我们可以使用这些类型构建一些新函数：

```
value :: Rank -> Integer
value Ten = 1
value Jack = 2
value Queen = 3
value King = 4
value Ace = 5
```

```
cardValue :: Card -> Integer
cardValue (rank, suit) = value rank
```

对任何card游戏来说，我们需要为card分派级别。这很简单，花色实际上并没有起到什么作用。我们简单地定义了一个计算级别值的函数，接下来定义另外一个计算cardValue的函数。下面是这个函数的实例：

```
*Main> :load cards-with-show.hs
[1 of 1] Compiling Main             ( cards-with-show.hs, interpreted )
Ok, modules loaded: Main.
*Main> cardValue (Ten, Hearts)
1
```

我们正在使用一个复杂的用户自定义类型元组。类型系统使得我们始终保持意图明确，这样一来就更容易推断出会发生什么了。

3. 函数与多态

早先，你见到过一些函数类型。看看下面这个简单的函数：

```
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

可以为这个函数增加一个类型，像下面这样：

```
backwards :: Hand -> Hand
...
```

这将限制backwards函数只能用于一种类型的列表，即card列表。我们真正想要的是下面这个函数：

```
backwards :: [a] -> [a]
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

现在，这个函数是多态的。[a]的含义是可以使用一个元素为任意类型的列表。这意味着可以定义一个接受某个类型a的列表作为参数并返回同样类型的列表的函数。我们使用[a] -> [a]构建了一个可以用于自定义函数的类型模板。通过这个声明告诉编译器如果传入一个整型数列表，这个函数将返回整型数列表。现在Haskell拥有足够的信息确保你保持诚信。

下面来构建一个多态数据类型。以下示例是一个多态类型，用于构建由类型相同元素组成的三元组：

haskell/triplet.hs

```
module Main where
    data Triplet a = Trio a a a deriving (Show)
```

在定义的左侧我们看到了data Triplet a。在这个实例中，a是一个类型变量。现在，任何由相同类型元素组成的三元组都具有类型Triplet a。看一个例子：

```
*Main> :load triplet.hs
[1 of 1] Compiling Main             ( triplet.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t Trio 'a' 'b' 'c'
Trio 'a' 'b' 'c' :: Triplet Char
```

我使用数据构造器Trio构建了一个三元组。在下一节中，我们将深入讨论数据构造器。基于类型声明，这个结果是Triplet a，或者更具体些，是Triplet Char，它可以满足任何一个以Triplet a为参数的函数。我们构建了一个完整的类型模板，用于描述任何三个类型相同的元素。

4. 递归类型

你也可以定义可递归的类型。例如，考虑一下树（tree）。你可以用多种方式定义树，不过在我们的树中，值都在叶子节点上。节点，要么是叶子，要么是树的列表。我们可以像下面这样来表达这棵树：

haskell/tree.hs

```
module Main where
    data Tree a = Children [Tree a] | Leaf a deriving (Show)
```

我们定义了一个类型构造器Tree和两个数据构造器Children和Leaf。将它们放在一起来表示树，如下所示：

```
Prelude> :load tree.hs
[1 of 1] Compiling Main             ( tree.hs, interpreted )
Ok, modules loaded: Main.
*Main> let leaf = Leaf 1
*Main> leaf
Leaf 1
```

首先，构建一棵只有一个叶子的树。将这个新叶子赋值给一个变量。数据构造器Leaf的唯一工作就是持有类型和值信息。我们可以通过模式匹配访问每个部分，像下面这样：

```
*Main> let (Leaf value) = leaf
*Main> value
1
```

接下来构建一些更复杂的树：

```
*Main> Children[Leaf 1, Leaf 2]
Children [Leaf 1,Leaf 2]
*Main> let tree = Children[Leaf 1, Children [Leaf 2, Leaf 3]]
*Main> tree
Children [Leaf 1,Children [Leaf 2,Leaf 3]]
```

我们构建了一棵包含两个子树（children）的树，每个子树都是一个叶子。接下来，构建了一棵包含两个节点的树，两个节点分别为叶子和一棵右子树。我们可以再次使用模式匹配获取每部分数据，可以从那里得到更复杂的数据。这个定义是递归的，所以可以通过let和模式匹配在所需要的更深的层次上获取数据。

8

```
*Main> let (Children ch) = tree
*Main> ch
[Leaf 1,Children [Leaf 2,Leaf 3]]
*Main> let (fst:tail) = ch
*Main> fst
Leaf 1
```

我们可以清晰地看出类型系统设计者的意图，能够剥离出完成工作所需要的数据。这个设计策略很明显将带来一些额外系统开销，不过当你深入研究抽象机制后，有时这些额外的开销也是值得的。在这个例子中，类型系统可以将函数附加到每个特定的类型构造器上。接下来看一个用于判定一棵树深度的函数：

```
depth (Leaf _) = 1
depth (Children c) = 1 + maximum (map depth c)
```

函数中的第一个模式很简单。如果它是一个叶子，那么无论叶子的内容是什么，树的深度都是1。

接下来的模式略有些复杂。如果在Children上执行depth，那么将在maximum (map depth c)上加1。函数maximum计算出一个数组中值最大的那个元素。你已经看到map depth c会得到一个由所有子树的深度值所组成的列表。在这里，你可以看到Haskell是如何使用数据构造器来帮助我们精确匹配出完成工作所需要的那部分数据结构的。

5. 类

到目前为止，我们已经学习了类型系统，知道了它在一些领域是如何工作的。我们构建了用户自定义类型构造器，并且获得了模板，这些模板能够定义数据类型并声明操作这些数据类型的函数。Haskell还有一个与类型相关的重要概念，而且是一个重量级的概念，这个概念称为类（class），不过请注意，因为不涉及数据，所以它不是面向对象编程中的那个类。在Haskell中，类可以精细地控制多态和重载。

例如，你无法将两个布尔值相加，但却可以将两个数字相加在一起。Haskell将类用作这个目的。具体地说，类定义了哪些操作可以在哪些输入上进行。你可以把它看作是一个Clojure 协议。

它是这样工作的。类提供了一些函数签名。如果类型支持类的所有函数，那么这个类型是类的一个实例。例如，在Haskell标准库中有一个名为Eq的类。

下面是这个类的定义：

```
class  Eq a  where
   (==), (/=) :: a -> a -> Bool

       -- 最小完整定义
       -- (==) 或 (/=)
   x /= y     =  not (x == y)
   x == y     =  not (x /= y)
```

这样，如果一个类型既支持==也支持/=，那么它是Eq的一个实例。你也可以指定一些样板实现。除此之外，如果实例只定义了其中的某个函数，其他函数将会被无偿提供。

类确实支持继承，并且它的运行和你想像中的一样。例如，类Num拥有子类Fractional和Real。图8-1中展示了Haskell98标准中最重要的一些类的组织结构图。记住，这些类的实例是类型，不是数据对象！

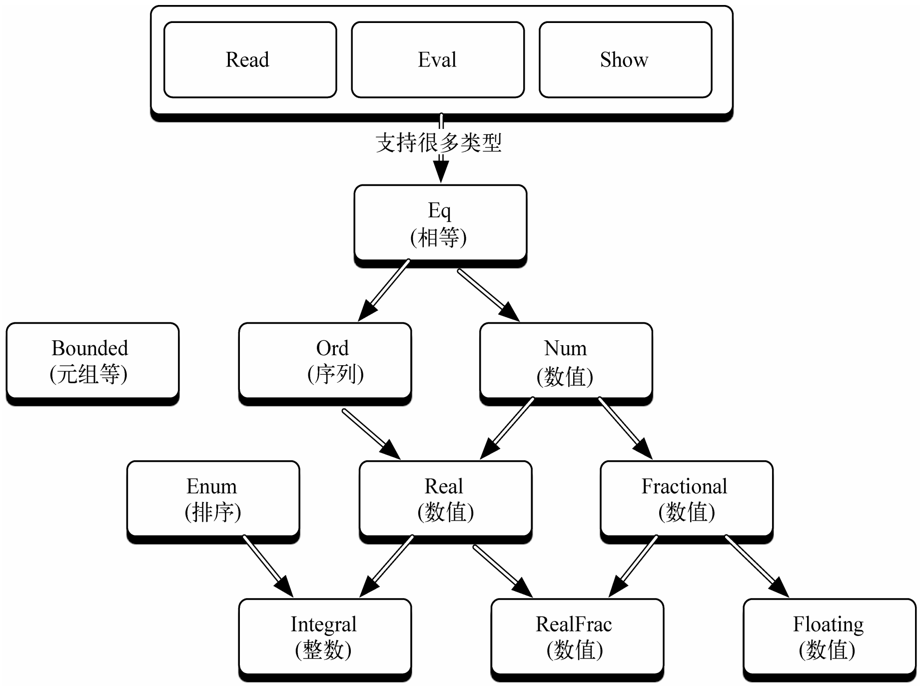


图8-1　重要的Haskell类

8.4.2　monad

自从我决定写这本书的那刻起，我就惧怕编写有关monad的章节。经过一些学习后，我了解到这个概念没有那么难。在这一节中，我将直观的告诉你需要monad的原因。接下来，我们会看到一些有关如何构建monad的高层次的描述。最后，我将介绍一些语法糖，它们应该可以真正地说明monad是如何工作的。

我依靠几本教程来帮助我形成自己的理解。我在Haskell Wiki[①]上读过一些实用的例子并且“Understanding monad”[②]（理解monad）这个页面上也有一些实用的好例子。不过要想更好地理解monad可以做些什么，你需要从大量不同的源码中阅读一些有关Monad的例子。

1. 问题：喝醉的海盗

比如，你知道一个海盗制作了一张藏宝图。他喝醉了，所以他选择了一个已知点和一个已知方向，蹒跚地连走（stagger）带爬（crawl），向着宝藏前进。一次走两步，爬一步。在命令式编程语言中，你会将语句连续地串在一起，其中v中存储着离原始起点的距离，如下所示：

```
def treasure_map(v)
    v = stagger(v)
```

① http://www.haskell.org/tutorial/monads.html。——原书注

② http://en.wikibooks.org/wiki/Haskell/Understanding_monads。——原书注

8

```
    v = stagger(v)
    v = crawl(v)
    return( v )
end
```

在treasure_map函数里，我们调用了多个函数，这些函数连续地改变状态——即移动的距离。这里的问题在于函数具有了可变的状态。我们可以使用函数式的方法解决这个问题，像下面这样：

haskell/drunken-pirate.hs

```
module Main where

    stagger :: (Num t) => t -> t
    stagger d = d + 2
    crawl d = d + 1
treasureMap d =
    crawl (
    stagger (
    stagger d))
```

你可以看到这个函数式的定义读起来很不方便。我们必须读作crawl、stagger和stagger，而不是stagger、stagger和crawl，而且这些参数放置的位置也很别扭。我们需要一种可以将一些函数连续地串联在一起的策略。我们可以用let表达式作为替代：

```
letTreasureMap (v, d) = let d1 = stagger d
                            d2 = stagger d1
                            d3 = crawl d2
                        in d3
```

Haskell可以将let表达式串联在一起，并且使用in语句表示最后的形式。你可以看到这个版本的实现几乎与第一个一样，仍旧无法让人满意。由于输入和输出是相同的，所以将这类函数组合起来应该更容易。我们想要将stagger(crawl(x))转化成stagger(x) · crawl(x)，其中·是一个函数组合。这就是一个monad。

总之，一个monad让我们可以以一种具有特定属性的方式组合函数。在Haskell中，我们将monad用于多种用途。首先，处理像I/O这样的事情很难，因为在纯函数式编程语言中，当传入相同的参数时，函数应该返回相同的结果，I/O除外，例如你想要函数基于一个文件内容的状态而改变。

同样，由于保存了状态，像前面介绍的有关喝醉海盗问题的代码也工作得很好。monad允许你模拟程序状态。Haskell提供了一种被称为do语法的特定语法支持命令式风格的编程。do语法工作时依赖monad。

最后，一些简单的诸如错误条件的事情在函数式语言中却难于处理，因为返回结果的类型根据函数成功与否而不同。Haskell提供了Maybe monad来解决这个问题。让我们再深入挖掘一下吧。

2. monad的组成

在基本级别上，一个monad包含以下三个基本的组成部分。

- 一个基于某容器类型的类型构造器。这个容器可以是简单的变量、列表或者是任何一个

可以持有值的对象。我们将使用这个容器来持有一个函数。你选择的容器将根据想要monad所做的事情的不同而不同。

- 一个名为return的函数，它负责将一个函数包装起来并放入容器。当后续使用到do记号时，这个名字会有意义。记住，return将函数包装到monad中。
- 一个名为>>=的bind函数，它负责给函数解包。我们会使用bind将函数串联在一起。

所有monad都需要满足三个规则。这里我会简单地介绍一下，对于某个monad m，某个函数f和某个值x而言：

- 你应该能够使用类型构造器创建monad，该monad可以与某个可以持有值的类型一同工作；
- 你应该能够在不损失信息的前提下对值进行包装和解包；(monad >>= return = monad)。
- 嵌套绑定函数应该与顺序调用它们具有相同的结果。((m >>= f) >>= g = m >>= (\x -> f x >>= g))。

我们不会在这些规则上花费过多时间，这些规则并不复杂。它们支持许多有用的信息无损转化。如果你真的想深入研究，请参见我给你留下的参考资料。

理论知识已经足够了。下面来构建一个简单的monad。我们将从头开始构建一个，接下来，将使用一些有用的monad来结束这个小节。

3. 从头构建一个monad

我们需要的第一件东西是一个类型构造器。monad将具有一个函数和一个值。像下面这样：

haskell/drunken-monad.hs

```
module Main where
    data Position t = Position t deriving (Show)

    stagger (Position d) = Position (d + 2)
    crawl (Position d) = Position (d + 1)

    rtn x = x
    x >>== f = f x
```

monad有三个主要的组成元素，包括类型容器、return和bind。这个monad可能是最简单的了。类型容器就是一个像data Position t = Position t的简单类型构造器。它唯一做的就是基于任意类型模板定义了一个基本类型。然后，我们需要利用return来将函数像值一样包装起来。由于monad很简单，所以仅需返回monad自己，使用(rtn x = x)很恰当地完成了包装。最后，我们需要一个允许组合函数的bind。这里的bind称为>>==，并且将它定义为只能调用与monad (x >>== f = f x)中的值相关的函数。我们使用>>==和rtn替代>>=和return以防止与Haskell内置的monad函数发生冲突。

注意，我们同时也重写了stagger和crawl，并使用自定义的monad替代了单纯的整数。我们可以拿这个monad试用一下。记住，我们找到了一种将嵌套转换为组合的语法。修订后的藏宝图代码：

8

```
treasureMap pos = pos >>==
                  stagger >>==
                  stagger >>==
                  crawl >>==
                  rtn
```

并且它正如我们期望那样的工作：

```
*Main> treasureMap (Position 0)
Position 5
```

4. monad和do记号

这种语法显然更好一些，不过你可以很容易想像到一些可进一步改进它的语法糖。Haskell的do语法恰恰就是用来做这个的。尤其是在解决I/O问题时，do语法能够派得上用场。在下面代码中，我们使用do记号从控制台读取一行文字并将该行文字反显出来。

haskell/io.hs

```
module Main where
    tryIo = do  putStr "Enter your name: " ;
                line <- getLine ;
                let { backwards = reverse line } ;
                return ("Hello. Your name backwards is " ++ backwards)
```

注意，程序一开始是一个函数声明。接下来，我们使用简单的do记法获得了符合monad的语法糖。这让程序感觉起来更像是有状态的和命令式的，不过我们实际上是在使用monad。你应该了解一些语法规则。

赋值用<-。在GHCI中，你必须用分号将代码行隔开并且用括号将do表达式和let表达式的正文括上。如果有多行代码，应该将每行代码都包裹在:{和}:中。现在，你终于看到调用monad的包装功能来构建return的原因了，它将一个返回值包装成一个do语法可以接受的整齐的形式。这些代码的行为好似一个有状态的命令式语言，不过它使用monad来管理有状态的交互。所有I/O操作都要紧密封装起来，并且必须都可以由do代码块中的某个I/O monad捕获。

5. 不同的计算策略

每个monad都有一个与计算相关的策略。我们在喝醉酒的海盗的例子中使用的身份（identity）monad只是将输入的东西回显。我们使用它将一个嵌套程序结构转换为顺序程序结构。让我们看看另外一个例子吧，这个例子看起来可能很奇怪，一个列表也是一个monad，其return和bind（>>=）函数的定义如下：

```
instance Monad [] where
    m >>= f  = concatMap f m
    return x = [x]
```

回顾一下， monad需要某个容器和一个类型构造器，还需要一个用于包装函数的return方法以及一个用于解包的bind方法。monad是一个类，[]将它实例化，从而获得了类型构造器。接下来我们需要类似return的用来包装结果的函数。

对于列表，我们将列表中的函数包装起来。要对其进行解包，bind针对列表中的每个元素调用函数，然后将得到的结果连接在一起。concat和map经常应用于序列，这实现了一个可以方便

地同时完成这两件事的函数，但也可以简单地使用concat (map f m)来实现。

想了解列表monad实际应用效果，看一下下面的使用do记号的代码：

```
Main> let cartesian (xs,ys) = do x <- xs; y <- ys; return (x,y)
Main> cartesian ([1..2], [3..4])
[(1,3),(1,4),(2,3),(2,4)]
```

我们使用do记号和monad创建了一个简单函数。从列表xs中得到x，从列表ys中得到y，然后返回x和y的每种组合。完成以上工作后，密码破解者程序实现起来就变得很简单了。

haskell/password.hs

```
module Main where
    crack = do x <- ['a'..'c'] ; y <- ['a'..'c'] ; z <- ['a'..'c'] ;
               let { password = [x, y, z] } ;
               if attempt password
                   then return (password, True)
                   else return (password, False)

    attempt pw = if pw == "cab" then True else False
```

这里，我们使用列表monad计算出了所有可能的组合。注意在这个上下文里，x <- [lst]的含义是“每个来自列表[lst]中的元素x”。我们让Haskell来完成这个困难的工作。此时，你需要做的只是尝试测试每个密码。密码被硬编码到attempt函数中了。有多种计算的策略可以用于解决类似列表推导的问题，不过这个问题展示了在列表monad背后的计算策略。

6. Maybe Monad

到目前为止，我们已经看到了身份monad和列表monad，通过后者，我们了解到monad的策略是支持一个主要的计算。在本小节中，我们来看看Maybe monad。我们将使用这个monad解决一个常见的编程问题：一些函数可能会返回失败。你可能会想到我们在讨论关于数据库和通信方面的问题，不过其他更简单的API同样需要支持失败的策略，考虑一个字符串搜索并返回字符串下标的例子。如果这样的字符串存在，那么返回结果类型为一个数字，否则则什么都不返回。

把这样的计算拼凑到一起是乏味单调的。这么来说吧，假设有一个解析网页的函数，你想得到HTML页、这个页的正文（body）以及正文的第一段内容，应该编写具有如下函数签名的函数：

```
paragraph XmlDoc -> XmlDoc
...

body XmlDoc -> XmlDoc
...

html XmlDoc -> XmlDoc
...
```

它们将支持一个类似下面的函数：

```
paragraph body (html doc)
```

问题是，paragraph、body以及html函数都有可能失败，所以需要支持一种可能是Nothing的类型。Haskell具有这样的类型，称为Just，Just x可以包装Nothing或者类似下面的一些类型：

```
Prelude> Just "some string"
Just "some string"
Prelude> Just Nothing
Just Nothing
```

你可以通过模式匹配去除Just。回到我们的例子中，paragraph、body和html文档都可以返回Just XmlDoc。然后，你可以使用Haskell的case语句（与Erlang的case语句行为相似）和模式匹配得到如下一些东西：

```
case (html doc) of
  Nothing -> Nothing
  Just x  -> case body x of
               Nothing -> Nothing
               Just y  -> paragraph 2 y
```

这个结果根本无法令人满意，考虑一下我们有可能这样编码paragraph 2 body (html doc)。我们真正需要的是Maybe monad。

下面是Maybe monad的定义：

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return         = Just
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x

    ...
```

我们包装的类型是一个类型构造器，即Maybe a。这个类型可以包装Nothing或者Just a。

return很容易，它只是包装了Just的结果，bind也很容易。对于Nothing，它返回一个返回结果为Nothing的函数。对于Just x，它返回一个返回结果为x的函数。两者都可以用return包装。现在，你可以将这些操作串联在一起：

```
Just someWebPage >>= html >>= body >>= paragraph >>= return
```

这样，我们就可以将这些元素完美地结合在一起了。它工作得很好，因为monad会负责决策需要组合的函数。

8.4.3　第三天我们学到了什么

在本节中，我们学习了三个难度较高的概念：Haskell类型、类以及monad。我们从类型开始，学习了现有函数的推断类型、数字类型、布尔类型以及字符类型，接下来我们进一步学习了用户自定义类型。我们举了一个基本例子，使用类型定义了由花色和级别组成的扑克牌。我们学到了如何参数化类型，甚至是使用递归类型定义。

接下来，我们将注意力完全集中于对monad的讨论。由于Haskell是一门纯函数式编程语言，所以它很难表达命令式风格的问题或当程序运行时状态累积的问题。Haskell的设计者依靠monad来解决这两个问题。monad是一个类型构造器，它包括了一些用于包装和将函数串联在一起的函数。你可以将使用不同类型容器的monad结合在一起，以支持多种不同的计算策略。我们使用

monad为程序提供了一种更加自然的命令式风格和处理多种可能性的方法。

8.4.4 第三天自习

找

- 一些有关monad的教程。
- 给出一个Haskell中monad的列表。

做

- 编写一个函数，该函数使用`Maybe` monad查找一个散列表值。编写一个散列表，该散列表可以多级存储其他的散列表。使用`Maybe` monad检索一个散列`key`对应的多级散列表中的元素。
- 用Haskell表示一个迷宫（Maze）。你将需要一个`Maze`类型、一个`Node`类型以及一个返回给定坐标节点的函数。这个节点应该具有一个到其他节点的出口列表。
- 使用一个列表monad解决迷宫问题。
- 在非函数式语言中实现一个monad（参见Ruby中的有关monad的文章系列[①]）。

8.5 趁热打铁

在本书的所有语言中，Haskell是唯一一个由委员会创建的语言。随着具有惰性机制的纯函数编程语言的影响力迅速扩大，一个委员会成立并致力于构建一个开放语言标准，可以整合加强现有函数语言能力并满足未来学术研究的需要。Haskell因此诞生，其1.0版本在1990年发布，自那时起这门语言及其社区一直成长至今。

Haskell支持各种函数式语言的特性，包括列表推导、惰性计算策略、偏应用函数和柯里化。默认状态下，Haskell函数每次仅处理一个参数，并使用柯里化支持多参数。

Haskell类型系统在类型安全和灵活性之间提供了良好的平衡。完全多态的模板系统为用户自定义类型和完全支持接口继承的类型类提供了丰富的支持。通常，除了在函数声明环节，Haskell程序员不用去了解类型的细节，而且类型系统还可以防止用户产生各种类型错误。

和使用其他纯函数编程语言一样，Haskell开发者在处理命令式风格程序和累积状态时必须富有创意。同时I/O也是一项挑战。幸运的是，Haskell开发者可以通过monad解决这些问题，monad是一个类型构造器和一个容器，后者支持将函数以值的形式包装和解包的基本函数，不同的容器类型提供不同的计算策略，这些函数允许程序员将monad以有趣的方式串联在一起。Haskell提供`do`语法，这个语法糖支持有一定约束的命令式风格语法。

8.5.1 核心优势

由于Haskell采用了完全的纯函数方法，没有任何妥协，因此其优势和不足之处也都十分明显。让我们逐一来说明吧。

① http://moonbase.rydia.net/mental/writings/programming/monads-in-ruby/00introduction.html。——原书注

1. 类型系统

如果你喜欢强类型（也许你可能不喜欢），你将喜欢上Haskell的类型系统，它召之即来，挥之即去。这个类型系统可以帮助我们避免常见错误，并且这些常见错误可以在编译阶段被捕获而不是运行阶段。不过安全性只是类型系统的一部分特性。

也许Haskell类型最具吸引力的地方就是，可以方便地将新行为与新类型关联到一起。你可以从头开始创建一个高级类型。使用类型构造器和类，你甚至可以毫不费力地定义出非常复杂的类型和类，诸如monad。有了类，自定义的新类型可以利用现有的Haskell标准库。

2. 表现力

Haskell语言的功能异乎寻常的强大。从抽象意义上说，它拥有用于简洁地表达强大功能概念的所有东西。这些概念包括了通过丰富的函数库和功能强大的语法所表现出的行为。那些概念还扩展到用于创建类型的数据类型中，甚至是无需过多语法即可将适当函数绑定到适当数据上的递归类型。在学术环境中，你再也找不到比Haskell更强大的函数式编程教学语言了，所有你需要的一切都在那里。

3. 编程模型的纯洁性

纯洁的编程模型可以从根本上改变你解决问题的方式，它会迫使你放下旧的编程范型去拥抱不同的做事方式。纯函数式编程语言赋予你可以依赖的东西。给定相同的输入，函数将总是返回相同的结果，这个属性使得程序推导变得更加容易。有时，你可以证明出程序是对的还是错的。你也可以避免许多因副作用而导致的错误，诸如意外的复杂性和不稳定，或者并发情况下的缓慢行为。

4. 惰性机制

曾几何时，用函数式编程语言编程就意味着使用递归。惰性计算机制提供一整套处理数据的新策略。你总是可以构建出表现更好的程序，并且与采用另外一种策略相比，代码行只是后者全部代码行的一小部分。

5. 学术支持

一些最重要、最有影响力的语言诸如Pascal成长于学术界，并受益于在学术环境下的研究和使用。作为函数式编程技术的主要教学语言，Haskell持续改善和成长。虽然它还不完全是一门主流编程语言，但你总是可以发现一些程序员使用它完成一些重要的工作。

8.5.2　不足之处

到目前为止，没有哪门语言可以完美地适合所有任务。Haskell的优势互补通常也有其不足的一面。

1. 编程模型不灵活

作为一门纯函数编程语言，Haskell提供了一些好处，不过同时也带来一些让你头疼的事情。你可能已经注意到了使用monad编程是本书关于编程语言的最后一章的最后一节，这是理所当然的。这些概念有着较高的能力要求。不过我们使用monad做了一些在其他语言中看起来微不足道的事情，诸如编写命令式风格的程序、处理I/O甚至是处理那些也许找到值也许没找到值的列表

函数。我之前谈论其他编程语言时曾提到过，这里我要再次重申一下。虽然Haskell把一些困难的事情变得简单了，但同时它也把一些简单的事情变得困难了。

特定的风格导致特定的编程范型。当构建一个逐步进行的算法时，命令式语言会工作得很好。函数式编程语言不适合I/O密集和脚本任务。在某个人眼中的纯洁性也许在另外一个人看来更像是一次失败的妥协。

2. 社区

说到妥协，你可以看到Scala和Haskell所采用方法的不同之处。虽然两者都是强类型的，但两者从根本上具备不同的设计哲学。Scala的一切都关于妥协，而Haskell的一切都关于纯洁。通过作出妥协，Scala在初始阶段就吸引了比Haskell更大的社区。虽然无法通过编程社区的大小来衡量语言的成功，但是要成功就必须拥有足够数量的支持者，并且拥有更多的用户可以提供更多的机会和社区资源。

3. 学习曲线

monad并非Haskell中唯一学习难度较高的概念。柯里化用于每个参数个数多于一个的函数中。大多数基本函数具有参数化的类型，并且应用于数字的函数常常使用类型类。虽然最终回报可能是值得的，但是你必须是一个拥有牢固理论基础的坚强程序员，这样你才能有机会在Haskell上获得成功。

8.5.3　最后思考

在本书所介绍的全部函数式编程语言中，Haskell是最难学的语言。把重点放在monad和类型系统上让学习曲线变得十分陡峭，不过一旦我掌握了其中一些关键概念，事情就变得容易许多，它也成为了我学过的回报最高的语言。基于类型系统以及monad应用的优雅，总有一天，你会回过头来看看这门本书中最重要的语言。

Haskell还扮演着另外一个角色。其方法的纯洁性和学术关注都将会提高我们对编程的认知。下一代最好的函数式编程程序员将会从Haskell中获得许多初步经验。

第 9 章

落幕时分

恭喜！你已经顺利完成了全部七门编程语言的学习。或许你现在有所期待，希望最后这章会将这些语言分出个高下，选出哪些是胜利者，哪些是失败者，然而，本书与胜败毫不相干。本书所要关注的是如何激发你的思想火花。如今的你，也许与职业生涯早期的我十分相似——同是某个缺乏想象力的庞大项目组中的一员，深陷于各种各样的商业项目的泥沼之中。与其说这是项目组，不如说是以机械化方式生产软件的工厂，纯把人当机器使。环境如此不堪，接触各种编程语言的机会自然少得可怜。那时的我，就好比某个酷爱电影的家伙，却居于偏远小镇，镇上只有一家电影院，放的还都是些所谓的“大片”。

直到我自立门户，开始自己生产软件时，我才真正领略到独立影片之妙。Ruby在我手中，简直就是出神入化。不过话说回来，我也没那么天真，会认为Ruby能解决一切问题。就像独立电影不断推动电影业发展那样，这些新兴的编程语言也在改变我们的组织以及编写程序的思维方式。下面我们就一起回顾一下，纵观全书，我们到底学到了什么。

9.1　编程模型

编程模型发展得极为缓慢。时至今日我们发现，大约每过20年，才会有一些新的编程模型涌现出来。回想我刚接触编程那会儿，我是从Basic和Fortran这样的过程式语言学起的，到了大学，通过学习Pascal，我明白了结构化编程是怎么回事，进入IBM之后，我开始用C和C++编写商业软件，之后，又初步认识了Java。这样一来，我踏进了面向对象编程的世界。我干编程这行三十多年，也只见过区区两种编程范型。你大概会奇怪，既然就见过两种，那干吗还积极介绍另外几种范型呢？嗯，这是个不错的问题。

这是因为，尽管编程范型发展很慢，但它的确在不断发展着。它就像肆虐的龙卷风，所到之处一片狼藉，只不过，它摧毁的是那些目光短浅的程序员和软件公司的前途。当你发觉自己正在为某种编程范型而内心纠结时，小心了，这可不是什么好信号。眼下，并发编程和可靠性编程都在一点点地改变着高级编程语言的发展方向。我认为，沿着当前的方向发展，即便最悲观的结果，也会有越来越多的解决特定问题的专用语言出现在我们面前。下面我们看看，本书都讲到了哪些编程范型。

9.1.1 面向对象（Ruby、Scala）

如今，面向对象无疑是最强势的编程范型，而Java正是面向对象语言的典型代表。这种范型有三大主要思想：封装、继承和多态。通过学习Ruby，我们知道了什么是鸭子类型。它不是用类或对象的定义施加某种类型契约，而是根据对象支持的方法确定类型。我们也了解到，通过代码块，Ruby能实现一些函数式编程思想。

Scala同样提供了面向对象编程范型。尽管它支持的是静态类型，然而，因其具有类型推断等用于简化语法的特征，因此比Java简洁得多。通过类型推断这一特征，Scala可根据其语法和用法中的蛛丝马迹，自动推导出变量类型。在引入函数式思想这方面，Scala甚至更胜Ruby一筹。

这两门语言现今都已广泛用于产品级应用当中。而且相比于Java这样的主流语言，它们都代表着语言设计的非凡进步。同时，面向对象语言也有不少变种。下面这种编程范型——原型语言正是其中之一。

9.1.2 原型编程（Io）

你或许认为，原型语言不过是面向对象语言的一个子集而已。然而，二者在编程实践当中的区别是非常明显的，因此我们将原型语言作为一种独立的编程模型加以介绍。和其他使用类来编程的语言不同，在原型语言中，所有原型都是对象实例，其中，某些实例经过特别设计，可用作其他对象实例的原型。原型语言家族的成员包括JavaScript和Io，它们既有简洁明了的形式，又有强大的表达能力，且通常是动态类型语言，因此在脚本开发、应用开发，尤其在用户界面等方面，表现都十分出色。

正像Io所展示的那样，即便只是简单的编程模型，但若始终保持语法的小巧、简洁，也同样能变得威力强大。我们已经把Io语言应用到了各种各样的环境当中，从脚本并发编程，到编写自己的DSL。但在本书中，原型编程还算不上最专用的范型。

9.1.3 约束-逻辑编程（Prolog）

Prolog出身于一个专用于约束－逻辑编程的语言家族。我们用Prolog编写各种应用，全是为了解决一小类问题。然而，解决这一小类问题获得的成果却蔚为壮观。简单地说，这类问题为某个已知问题域定义一些逻辑约束，然后就可以用Prolog求出这类问题的解。

要是编程模型符合这种范型，我们就能仅用一小段代码，完成其他语言写好多行代码才能做到的事。这一语言家族打造了许多当今世界至关重要的应用，比如航空交通管制、土木工程等。在其他语言如C和Java当中，也有一些粗糙的逻辑规则引擎。此外，Prolog还是Erlang的灵感之源。说到Erlang，它来自于本书介绍的另一大语言家族。

9.1.4 函数式编程（Scala、Erlang、Clojure、Haskell）

函数式编程或许是本书寄予厚望的编程范型。虽然在函数式语言里，有的惟精惟一，有的驳

杂不纯，但它们都蕴含着同一思想。函数式程序由数学函数构成，调用同一个函数，都会返回同样的结果，尽可能地避免副作用，甚至严格禁止。至于如何写这些函数，方式可以多种多样。

你已经了解到，函数式编程语言通常比面向对象语言有更强的表达能力。用它们写的代码显得比面向对象语言更短小精悍，因为它们用来编程的手段要丰富得多。我们介绍了高阶函数，还有柯里化这样的复杂概念，而这些概念，在面向对象语言中是难得一见的。我们在Haskell那一章学到过，函数式的纯正程度会产生不同的优缺点。对函数式语言来说，避免副作用是显而易见的优势，这能使并发编程不再棘手。一旦可变状态的阴霾烟消云散，许多传统的并发问题也就迎刃而解了。

9.1.5　范型演进之路

如果你下定决心，以后多用函数式语言来编程，那么有几条路可供你选择，你既可以彻底与OOP决裂，也可以选择较为温和的渐进策略。

我们学这七门语言花了不少心力，但也见识到了叱咤风云四十余载的几门语言，以及多种编程范型。但愿你现在已经对编程语言的演化有了一个正确认识。你可以看到三条截然不同的范型演进之路。对Scala而言，其路线是和谐共处。Scala程序员可以用浓重的函数式风格写出面向对象程序。Scala这门语言最根本的性质，就是两种编程范型平起平坐。Clojure走的路子是兼收并蓄。Clojure搭建在JVM之上，它的应用程序能直接使用Java对象。但Clojure的理念认为OOP的某些基本元素具有根本缺陷。因此，与Scala不同的是，Clojure通过Clojure-Java互操作（Clojure-Java Interop）去利用Java虚拟机上的现有框架，而不是去扩展Java语言本身。Haskell和Erlang差不多算是独立语言。从思想上说，它们在任何形式上都和面向对象编程不沾边。综上所述，你既可以同时采用两种范型，也可以和面向对象彻底决裂，还可以骑驴找马——先用着面向对象库，以后再决定是否抛弃OOP范型。这三条路任由你选择。

无论是否选用本书介绍的语言，你都能比较透彻地了解面前的选择。作为悲催的Java开发者，我不得不为闭包苦苦等上十年，只因像我这样热切期盼闭包的人都是“文盲”或“半文盲”，没法给闭包摇旗呐喊、鼓吹造势，也因为Spring这类主流框架，坚持用匿名内部类来解决大量本可用闭包解决的问题。没有闭包，输入代码的工作就太繁重了，我的手指头都敲出血了；阅读代码的任务也不轻，我的双眼也布满了红红的血丝。现在的Java程序员再不会像从前那样一无所知、任人宰割，因为Martin Odersky和Rich Hickey这样的人为我们提供了众多其他选择，这些选择不仅推动编程语言水平的不断提高，也逼迫Java或不断进步、或被潮流淘汰。

9.2　并发

本书反复出现的主题，就是找到能更好地处理并发的语言结构和编程模型。本书所介绍的这些语言，在处理并发的方法上各有不同，但相同的是，它们处理并发都极为有效。让我们再来浏览一遍之前见过的这些方法。

9.2.1 控制可变状态

迄今为止，关于并发的讨论中最常见的主题就是编程模型。面向对象编程会导致副作用和可变状态。若综合考虑这两个因素，程序就会变得异常复杂。当多个线程和进程共同使用时，复杂性会高到难以控制。

函数式编程语言增加了一条重要规则，也因此增加了结构。多次触发同一函数将产生相同结果。变量是单赋值的。消除了副作用，竞争条件和一切与副作用相关的复杂性也就随之消除。然而，我们还见到了基本编程模型之外的一些切实有效的技术。下面，我们就来仔细看看这些技术。

9.2.2 Io、Erlang 和 Scala 中的 actor

无论用对象还是进程，actor方法都始终如一。它获取从对象内部发出的非结构化的进程间通信，将其转化为头等结构之间的消息传递，且每个actor都拥有一个消息队列。Erlang和Scala语言使用模式匹配，对传递进来的消息进行匹配，然后根据条件执行它们。在第6章，我们举了个俄罗斯轮盘赌的例子，来说明什么是濒死进程。还记得吧，我们把弹子放在了第三格中：

erlang/roulette.erl

```
-module(roulette).
-export([loop/0]).

% send a number, 1-6
loop() ->
    receive
        3 -> io:format("bang.~n"), exit({roulette,die,at,erlang:time()});
        _ -> io:format("click~n"), loop()
end.
```

然后，我们启动一个进程Gun，并赋给它ID。我们可以用`Gun ! 3`终止该进程。Erlang的语言和虚拟机支持健壮的监控，可以在出现问题的第一时间通知用户，甚至重启进程。

9.2.3 future

除actor之外，Io还添加了两种并发结构：协程和future。协程可让两个对象合作处理多任务，而它们又能各自选择适当时机放弃控制权。回想一下，future就好比长时运行的并发计算的占位符。

我们执行过一条语句`futureResult := URL with("http://google.com/") @fetch`。虽然它无法立即获得结果，但程序控制权却能马上回到我们手中。只有当我们试图访问future时，程序才会阻塞。到产生结果的时候，Io的future会自动转化为该结果。

9.2.4 事务型内存

在Clojure中，我们见识到了一些有趣的并发处理方法。软件事务型内存（Software

Transactional Memory，STM）包装了事务中某个共享资源的每一个分布式访问。相同的方法也可用于数据库对象，能在触发并发操作时保持数据库的完整性。我们在dosync函数中封装了每一个访问。用了这种方法，Clojure开发者就能摆脱严格的函数式设计，写出合理的代码，同时在多个线程和进程间保持完整性。

STM是一种相对来说较新的思想，正逐渐被越来越多的流行语言所采用。因为Lisp是一门多范型语言，所以Clojure作为Lisp的传人，用起STM来也是得心应手。当用户确信即使处在高并发访问的条件下，应用程序也能保持完整性和高性能，那么，他们就能放心使用各种不同的编程范型了。

下一代程序员会对他手中的编程语言提出更多要求。光是给个方向盘和变速杆，能发动线程，并在出现信号灯时停下来等待，这些已经满足不了他们了。新式语言必须有条理清晰、前后一致的并发处理思想，还要有与之配套的一组方法。或许这样的并发需求会使现有的编程范型全都过时，但它同样可能督促较老的语言与时俱进，使其对可变状态采用更严格的控制措施，并采用actor和future这样更出色的并发结构。

9.3　编程结构

写这本书最激动人心的事，就是把书中各门语言的基本元素展现在读者面前。每当我介绍一门新语言，都会将其与众不同的新思想进行一番讲解。下面是你探索其他语言时很可能遇到的一些编程结构。同时，它们也是我探索语言得到的最宝贵的财富。

9.3.1　列表解析

正如你在Erlang、Clojure和Haskell[①]中看到的那样，列表解析是一种简洁而紧凑的结构，它在结构中融合了筛选器、映射、笛卡儿积等几种概念，使这种结构变得非常强大。

第一次遇见列表解析，是在介绍Erlang那章。我们先是用Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}]这样的列表，表示添加了商品的购物车。然后，如果想为列表加上税款元素，只需用一条列表解析语句，问题马上就能迎刃而解。如下所示：

```
8>  WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
8>  {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

几位语言发明者都不约而同地提到，列表解析是他们最钟爱的特性之一。对他们这一见解，我亦是深表赞同。

9.3.2　monad

要说写这本书让我在哪个领域的知识增长最多，那可能还得是monad。在纯正的函数式语言

① Scala同样支持列表解析，只是我们没有用到它们。——原书注

中，我们不能用可变状态编程，但我们可以编写monad，monad有助于对问题进行结构化，让我们写函数时感觉可变状态可用。Haskell中具有monad特性的do符号，就是用来解决可变状态这一问题的。

我们还发现，monad能简化复杂计算。每一个单子都提供了一种计算策略。我们使用Maybe monad处理失败条件，比如，可能返回Nothing的列表搜索。此外，我们还用List monad计算笛卡儿积并破解组合数。

9.3.3　匹配

我们在书中见过的更为常用的一种编程特性是模式匹配。我们首次见到这种编程结构是在Prolog中，但Scala、Erlang、Clojure和Haskell中也出现过。这些语言都借助模式匹配的力量，极大地简化了代码。模式匹配可解决的问题包括语法分析、分布式消息传递、解构、合一、XML处理，等等。

说到典型的Erlang模式匹配，请回想一下Erlang那章实现过的翻译服务：

erlang/translate_service.erl

```
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
        {From, "casa"} ->
            From ! "house",
            loop();

        {From, "blanca"} ->
            From ! "white",
            loop();

        {From, _} ->
            From ! "I don't understand.",
            loop()

end.


translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

这个循环函数匹配后接单词（casa或blanca）或通配符的进程ID（From）。这一模式匹配无需借助程序员编写的语法分析，就能快速提取消息中的重要部分。

9.3.4　合一

Prolog用到了合一，合一和模式匹配的关系，就像表兄弟一样。你已经学过，Prolog可将合法值替换到规则内，从而使规则的左右两边匹配。Prolog会尝试不同的值，直到穷尽所有合法值为止。我们在Prolog那章见过一个简单的程序concatenate，以作为合一的示例：

prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
  concatenate(Tail1, List, Tail2).
```

我们知道，合一能让程序威力大增，因为它有三大功效：测试真值、匹配左端、匹配右端。

9.4　发现自己的旋律

整本书中，我们谈论了很多电影和电影角色，其实，拍电影的乐趣就在于把自己的经历和体验融入到演员、场地、外景之中，让它们来讲述你想要讲述的故事。你所做的一切都是为了取悦观众。你知道得越多，拍出来的电影也就越精彩。

我们需要以同样方式思考编程。和电影一样，程序也有观众。不过，我说的观众并不是使用应用程序的用户，而是阅读代码的人。想成为一名伟大的程序员，你必须为你的观众编写代码，找到愉悦他们的独特旋律。如果你明白其他语言都提供了哪些特性，那你就有更大的空间来发现这种旋律，并能不断对其加以精炼和改进。你通过代码表达自我的方式，就是你独一无二的旋律。而这旋律，一定来自于你的亲身经历与体验。我希望这本书能帮你找到自己的旋律，最重要的是，希望你乐在其中。

附录

参考书目

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2007.

[Gra04] Paul Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly & Associates, Inc, Sebastopol, CA, 2004.

[Hal09] Stuart Halloway. *Programming Clojure*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, Inc., Mountain View, CA, 2008.

[TFH08] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide.* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

版 权 声 明

Seven Languages in Seven Weeks
A Pragmatic Guide to Learning Programming Languages

七周七语言 理解多种编程范型

“为了了解更多的范例，提高编程水平，我读了很多相关的技术书，其中不乏让人拍案叫绝之作，本书即是一例。Bruce有丰富的学习经历和使用多种语言的经验，他把一些杰出的范例很好地组合到了一起，让读者从他的这些经验中有所斩获。我极力推荐！”

——Vankat Subramaniam博士，Agile Developer公司创始人，敏捷开发权威

“看过这本书后，我解决问题的思路扩宽了，学习新语言的热情也被点燃了。”

——Travis Kaspar，Northrop Grumman公司软件工程师

从计算机发展史早期的Cobol、Fortran到后来的C、Java，编程语言的家族不断壮大。除了这些广为人知的语言外，还涌现了Erlang、Ruby等后起之秀，它们虽被喻为小众语言，但因其独特性也吸引了为数不少的追随者。

Bruce A. Tate是软件行业的一名老兵，他有一个宏伟目标：用一本书的篇幅切中要害地探索七种不同的语言。本书就是他的成果。书中介绍了Ruby、Io、Prolog、Scala、Erlang、Clojure和Haskell这七种语言，关注每一门语言的精髓和特性，重点解决如下问题：这门语言的类型模型是什么，编程范型是什么，如何与其交互，有哪些决策构造和核心数据结构，有哪些独特的核心特性。

在这个飞速发展的信息时代，程序员仅仅掌握甚至精通一门语言是远远不够的。了解多门语言蕴涵的思维方式，在编码中互相借鉴，再挑出一两门对自己口味的语言深入学习，这些已经成为在软件行业中安身立命之本。从这个意义上说，每个程序员都应该看看这本《七周七语言》。

The Pragmatic Programmers

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-27611-7

ISBN 978-7-115-27611-7
定价：69.00元